

SmartMesh IP User's Guide

Table of Contents

1	About This Guide	5
1.1	Related Documents	5
1.1.1	Getting Started with a Starter Kit	5
1.1.2	User's Guide	5
1.1.3	Interfaces for Interaction with a Device	5
1.1.4	Access Point Notes	6
1.1.5	Software Development Tools	6
1.1.6	Application Notes	6
1.1.7	Documents Useful When Starting a New Design	6
1.1.8	Software	7
1.1.9	Other Useful Documents	7
1.2	Conventions Used	8
1.3	Revision History	9
2	The SmartMesh IP Network	10
2.1	Introduction	10
2.1.1	About SmartMesh IP	10
2.1.2	Network Overview	11
2.2	Network Formation	16
2.2.1	Mote ID	17
2.3	Bandwidth and Latency	17
2.3.1	NumParents Parameter	18
2.3.2	BWMult Parameter	18
2.3.3	BaseBW Parameter	19
2.3.4	Services	19
2.3.5	Cascading Links	19
2.3.6	Normal/Fast Downstream Modes	20
2.4	Data Traffic	20
2.4.1	Directly Connected "Gateway" Application	21
2.4.2	End-to-End IP	22
2.5	Network Security	23
2.5.1	Security Layers	23
2.5.2	Security Levels	24
2.6	Blink Feature	25
2.6.1	Introduction	25
2.6.2	When to use Blink	26
2.6.3	Detailed Description	27
2.6.4	Manager Requirements	30
3	The SmartMesh IP Mote	31
3.1	Introduction	31

3.1.1	Steps in a Mote Design	31
3.2	Mote State Machine	31
3.3	Joining	33
3.3.1	Programmatic Join	33
3.3.2	Auto Join	34
3.3.3	Joining Adjacent Network	35
3.4	Services	35
3.4.1	Requesting Bandwidth	35
3.4.2	Back-Off Mechanism	36
3.5	Communication	37
3.5.1	Sockets	37
3.5.2	UDP Port Assignment	37
3.5.3	Sending and Receiving Data	38
3.6	Events and Alarms	38
3.7	Factory Default Settings	39
3.8	Power Considerations	39
3.8.1	Power Source Information	39
3.8.2	Routing Mode	40
3.8.3	Join Duty Cycle	40
3.9	Master vs. Slave	41
3.9.1	Modes	41
3.9.2	LEDs	41
3.9.3	Master Behavior	41
3.9.4	Switching To Slave Mode	42
3.9.5	Switching To Master Mode	42
3.10	Over the Air Programming (OTAP)	43
4	The SmartMesh IP Managers	44
4.1	The SmartMesh IP Embedded Manager	44
4.1.1	Introduction	44
4.1.2	Accessing the Manager	45
4.1.3	Configuration and Usage	47
4.1.4	Network Activity	49
4.1.5	Access Control	51
4.1.6	Restoring Manager Factory Default Settings	52
4.1.7	Channel Blacklisting	53
4.2	The SmartMesh IP VManager	54
4.2.1	Introduction	54
4.2.2	Access Point Notes	57
4.2.3	Installation	59
4.2.4	Interfacing to the Manager	70
4.2.5	Configuration and Usage	72
4.2.6	Network Activity	74
4.2.7	Communicating with Motes	76

4.2.8	Access Control	78
4.2.9	Redundancy	79
4.2.10	Over the Air Programming (OTAP)	80
4.2.11	Restoring Manager Factory Default Settings	81
4.2.12	Channel Blacklisting	82
4.2.13	Database Backups and Restores	83
5	SmartMesh Glossary	84

1 About This Guide

1.1 Related Documents

The following documents are available for the SmartMesh IP network:

1.1.1 Getting Started with a Starter Kit

- [SmartMesh VManager Easy Start Guide](#) - walks you through basic VManager installation and a few tests to make sure your network is working.
- [SmartMesh IP Embedded Manager Easy Start Guide](#) - walks you through basic embedded manager installation and a few tests to make sure your network is working.
- [SmartMesh IP Embedded Manager Tools Guide](#) - the installation section contains instructions for installing the serial drivers and example programs used in the Easy Start Guide and other tutorials.

1.1.2 User's Guide

- [SmartMesh IP User's Guide](#) - describes network concepts, and discusses how to drive mote and manager APIs to perform specific tasks, e.g. to send data or collect statistics. This document provides context for the API guides. It also contains a glossary of SmartMesh terms.

1.1.3 Interfaces for Interaction with a Device

There are two interfaces for interaction with a Manager - an Application Programming Interface (API) for programmatic interaction, and a Command Line Interface (CLI) for human interaction.

- [SmartMesh IP Embedded Manager CLI Guide](#) - used for human interaction with an embedded manager (e.g. during development of a client, or for troubleshooting). This document covers connecting to the CLI and its command set.
- [SmartMesh IP Embedded Manager API Guide](#) - used for programmatic interaction with an embedded manager. This document covers connecting to the API and its command set.
- [SmartMesh IP VManager CLI Guide](#) - used for human interaction with a VManager (e.g. during development of a client, or for troubleshooting). This document covers connecting to the CLI and its command set.
- [SmartMesh IP VManager API Guide](#) - used for programmatic interaction with a VManager. This document covers connecting to the API and its command set.
- [SmartMesh IP Mote CLI Guide](#) - used for human interaction with a mote (e.g. during development of a sensor application, or for troubleshooting). This document covers connecting to the CLI and its command set.

- [SmartMesh IP Mote API Guide](#) - used for programmatic interaction with a mote. This document covers connecting to the API and its command set.

1.1.4 Access Point Notes

- [SmartMesh IP User's Guide](#) - describes reprogramming DC2274 for use as an Access Point Mote.
- [VManager AP Bridge User's Guide](#) - user's guide for the Access Point Bridge reference software

1.1.5 Software Development Tools

- [Dustcloud.org](#) - contains documentation and links to various open source software tools for exercising mote and manager APIs and visualizing the network.

1.1.6 Application Notes

- [SmartMesh IP Application Notes](#) - Cover a wide range of topics specific to SmartMesh IP networks and topics that apply to SmartMesh networks in general.

1.1.7 Documents Useful When Starting a New Design

- The Datasheet for the mote being used, e.g. the [LTC5800-IPM SoC](#), or one of the [modules](#) based on it.
- The Datasheet for the embedded manager being used, e.g. the [LTC5800-IPR SoC](#), or one of the [embedded managers](#) based on it.
- A [Hardware Integration Guide](#) for the mote/manager SoC or [module](#) - this discusses best practices for integrating the SoC or module into your design.
- A [Hardware Integration Guide](#) for the embedded manager - this discusses best practices for integrating the embedded manager into your design.
- A [Board Specific Integration Guide](#) - For SoC motes and Managers. Discusses how to set default IO configuration and crystal calibration information via a "fuse table".
- [Hardware Integration Application Notes](#) - contains an SoC design checklist, antenna selection guide, etc.
- The [ESP Programmer Guide](#) - a guide to the DC9010 Programmer Board and ESP software used to load firmware on a device.

1.1.8 Software

- ESP software - used to program firmware images onto a mote or module. Described in the [ESP Programmer Guide](#).
- Fuse Table software - used to construct the fuse table as discussed in the [Board Specific Configuration Guide](#).

1.1.9 Other Useful Documents

- A list of [Frequently Asked Questions](#).


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, menu commands, and device states and modes.

Italic type is used to introduce a new term, and to refer to APIs and their parameters.

 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`

1.3 Revision History

Revision	Date	Description
1	07/17/2012	Initial release
2	03/18/2013	Numerous small changes
3	10/22/2013	Minor corrections
4	04/04/2014	Details on the temperature data generator for master mode
5	10/28/2014	Clarified blacklisting requirements; Other minor corrections
6	04/22/2015	Deprecated autostart command; Updated blacklisting requirements; Added description of OCSDK
7	12/03/2015	Deprecated software licencing commands; Clarified description of restore command; Other minor corrections
8	08/19/2016	Greatly modified to incorporate VManager; Phase I production
9	11/04/2016	More changes for VManager; Added section on Blink Feature
10	03/15/2017	Updates for VManager 1.1.0

2 The SmartMesh IP Network

2.1 Introduction

The Network User's Guide is intended to explain fundamental network and device behavior at a high level. All SmartMesh IP hardware includes the necessary compiled firmware to enable reliable mesh communication. Network and device behavior is configured programmatically via an Application Programming Interface (API). For additional details on the APIs referenced in this document, see the [SmartMesh IP Embedded Manager API Guide](#), the [SmartMesh IP VManager API Guide](#), or the [SmartMesh IP Mote API Guide](#).

2.1.1 About SmartMesh IP

SmartMesh IP combines reliability and ultra low-power with a native Internet Protocol (IP) layer for a robust, standards-based offering perfect for a broad range of applications. SmartMesh IP provides robust wire-free connectivity for applications where low power, reliability, and ease of deployment matter. With Linear's Eterna™ 802.15.4e SoC technology, every node in a SmartMesh IP network can run on batteries for years, or virtually forever with an energy harvesting power source. The ability to put a sensor anywhere, without wiring, allows network deployments that do not disrupt building operations or occupants. With built-in intelligence that enables the mesh network to self-form and self-maintain, SmartMesh IP systems are easily deployed by field technicians with no wireless technology expertise.

A SmartMesh IP network consists of the following;

- A SmartMesh IP manager. Two manager options are available:
 - An embedded version which runs on the SmartMesh Eterna SoC typically used for small (less than 100 devices) networks with an integrated Access Point Mote (APM)
 - A larger server-class version called VManager for larger installations, and can be used with multiple Access Point Motes.
- Network devices which can take the form of:
 - Default factory firmware mote in **master** mode (no external API client required) - see [Master vs. Slave](#) for a description of the two modes
 - Default factory firmware mote in **slave** mode attached to an API client such as a microcontroller.
 - Mote built with a customer application using the [On-Chip Software Development Kit \(OCSDK\)](#)

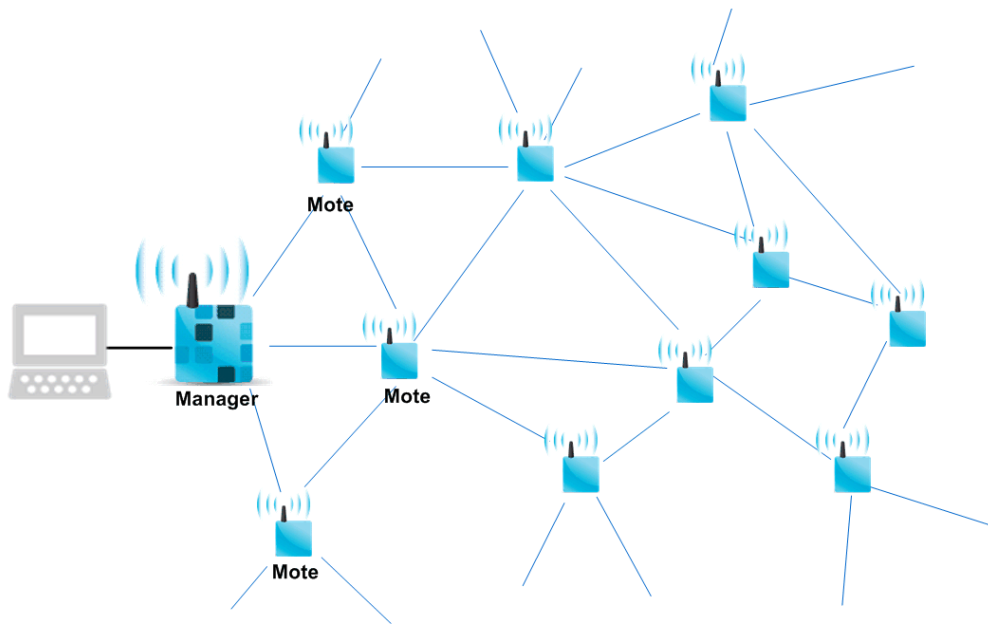


Figure 1 - SmartMesh IP Network

SmartMesh IP Features

SmartMesh IP system features include:

- **Ultra low-power network** - The network can run on batteries, energy harvesting, or line power
- **High network reliability** - >99.999% network reliability even in harsh RF environments
- **IPv6 addressability** - Combines 6LoWPAN with IEEE 802.15.4e
- **Comprehensive security management** - Allows you to configure NIST-certified AES-128 based security to meet your requirements
- **Flexible configuration** - Network parameters can be selected to match specific system requirements (power / latency / bandwidth)
- **Fully tested network stack and manager software** - Application programming interfaces are used to communicate with and configuring the product - no user networking code necessary.

2.1.2 Network Overview

A SmartMesh® network consists of a self-forming multi-hop mesh of nodes known as *motes* (1), and an *Access Point mote* (2) that connects the motes to the *Network Manager* (3) that monitors and manages network performance and security, and acts as a bridge between the host application and the wireless network. Motes are capable of two way communication and collect and relay data.

SmartMesh networks communicate using a Time Slotted Channel Hopping (TSCH) link layer, pioneered by Linear's Dust Networks group. In a TSCH network, all motes are precisely synchronized to within tens of microseconds. Time in the network is organized into timeslots, which enables collision-free packet exchange and per-transmission channel-hopping. In a SmartMesh network, every device has one or more *parents* (e.g. mote 3 has motes 1 and 2 as parents) that provide redundant *paths* to overcome communications interruption due to interference, physical obstruction or multi-path fading. If a packet transmission fails on one path, the next re-transmission may try on a different path and different RF channel. Building networks with sufficient redundancy requires following some simple deployment guidelines - these are outlined in the [application note](#) "Planning a Deployment."

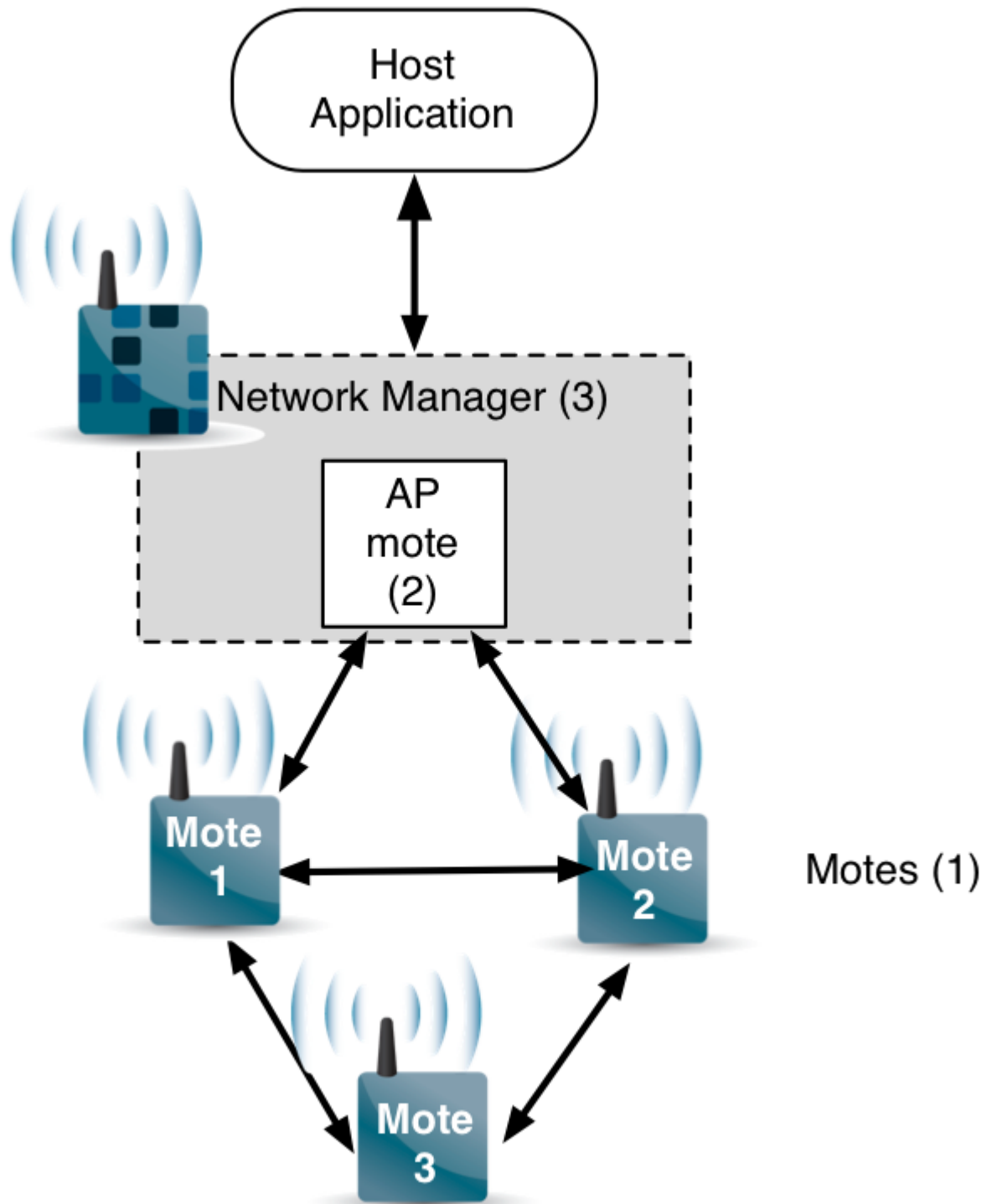


Figure 2 - A Simple Network

A network begins to form when the Network Manager in Figure 2 instructs the access point mote(s) (AP) to begin sending *advertisements* - packets that contain information that enables a device to synchronize to the network and request to join. This message exchange is part of the security handshake that establishes encrypted communications between the manager or application, and mote. Once motes have joined the network, they maintain precise synchronization through time correction messages sent between connected neighbors.

An ongoing *discovery* process ensures that the network continually discovers new paths as the RF conditions change. In addition, each mote in the network tracks performance statistics (*e.g.* quality of used paths, and lists of potential paths) and periodically sends that information to the network manager in packets called *health reports*. The Network Manager uses health reports to continually optimize the network to maintain >99.999% data reliability even in the most challenging RF environments.

The use of TSCH allows SmartMesh devices to sleep in-between scheduled communications and draw very little power in this state. Motes are only active in timeslots where they are scheduled to transmit or receive, typically resulting in a duty cycle of <1%. The optimization software provided with the Network Manager coordinates this schedule automatically.

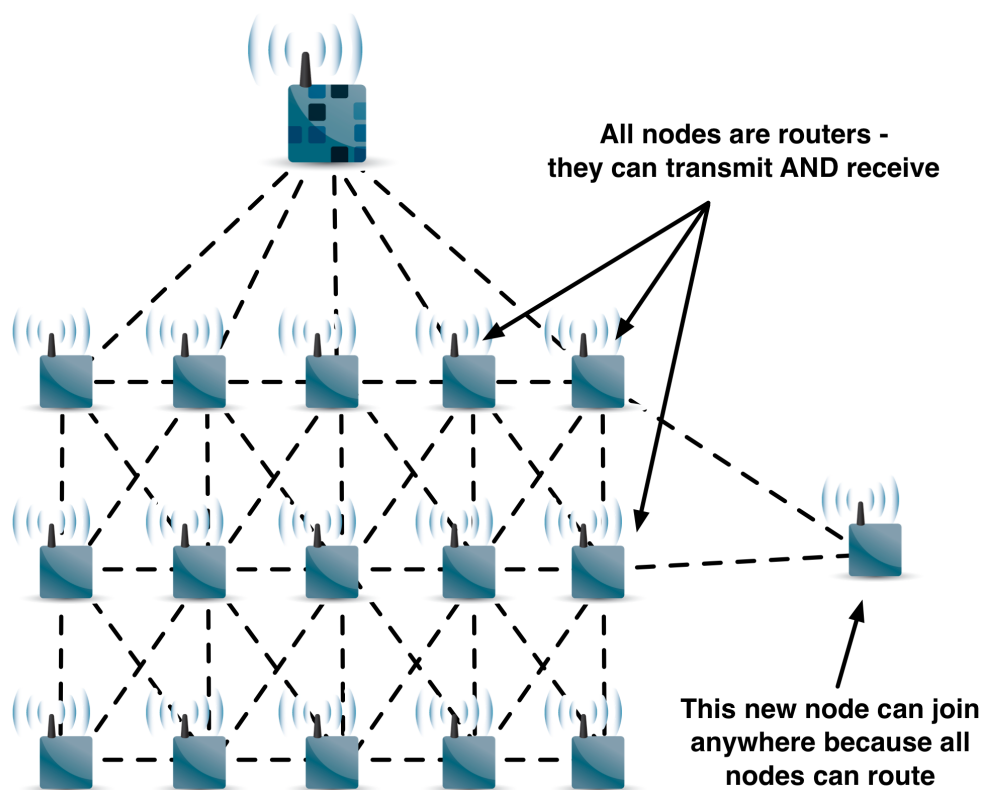


Figure 3- All Nodes are Routers

The combination of very low duty cycle and the Eterna low-power radio allows every mote in a SmartMesh network – even busy routing ones – to run on batteries for years. By default, all motes in a network are capable of routing traffic from other motes, as shown in Figure 3, which simplifies installation by avoiding the complexity of having distinct routers vs. non-routing end nodes. Motes may be configured as *non-routing* to further reduce that particular mote’s power consumption where ultra low power is a must, *e.g.* with energy harvesting devices.

At the heart of SmartMesh motes and AP motes is the Eterna IEEE 802.15.4e System-on-Chip (SoC), featuring our highly-integrated, low power radio design, plus an ARM® Cortex™-M3 32-bit microprocessor running SmartMesh networking software. The SmartMesh software comes fully compiled yet is configurable via a rich set of application programming interfaces (APIs) which allows a host application to interact with the network, e.g. to transfer information to a device, to configure data publishing rates on one or more motes, or to monitor network state or performance metrics. Data publishing need not be uniform, each device is free to publish as frequently or infrequently as is required for that application.

2.2 Network Formation

For a mote to join a network, it must get time-synchronized to other devices by hearing an *advertisement* from an Access Point (AP) mote or a mote already in the network. The network starts forming when the manager instructs the AP(s) to begin sending advertisements. One mote will hear the AP advertisement, then join, and start advertising itself. This process repeats in parallel as other motes join and begin their own advertising. The advertisements are IEEE 802.15.4e Beacon frames that contain synchronization and link information. In addition to synchronizing the new device, the advertisement also describes when the new device can send in a request to join the network, and when it should expect a reply. This results in temporary shared links being assigned to the joining mote that it will use until it gets its specific dedicated links from the manager.

By default, SmartMesh IP APs advertise four times per 256-slot slotframe, on average approximately once every 500 ms. The first motes to join the network after hearing one of these advertisements will do so according to the following formula, assuming the mote join duty cycle is set to 5% and a typical 80% path stability. A faster join time can be achieved by increasing the mote's join duty cycle at the expense of higher average power during the search process.

```
Synch time = adv interval per device * #channels / (#advertisers * path stability * join duty cycle)
            = 0.5 s * 15 / (1 * 0.8 * 0.05)
            = 188 s
```

Mote join duty cycle is configurable by setting the *joinDutyCycle* mote parameter - at 100% duty cycle the synch time falls to < 10 s. Setting a higher join duty cycle increases the power of the searching mote but allows it to synch up more quickly. If your network has a lot of 1-hop motes though, a low join duty cycle might not slow down the total join time significantly, since there are more advertisers and there are many motes joining in parallel, so the synchronization time for an individual mote is small relative to the total network formation time. For applications with ultra-low power limits, the join duty cycle can be set as low as 0.5%.

After synchronizing, the mote waits for one slotframe while continuing to listen with the goal of hearing more advertisements from other devices. After this additional listening, the joining mote sends in a *join request* consisting of power source and routing-capability information, as well as a list of heard neighbors. This message exchange is part of the [security](#) handshake that establishes encrypted communications between the manager or application, and mote, and is encrypted using a shared secret *join key*. The manager responds with a run-time MAC layer authentication key, a session to the manager, a short address to be used by the mote for all communications from this point on, and a route to the manager. Additional packets are exchanged to establish a broadcast session and associated route, slotframe and link assignment, setting of a time parent, and an explicit activate command which tells the mote to switch from temporary links to those explicitly added.

At this point the device may request additional bandwidth for publishing data. A mote that is fully joined into the network will also be advertising, as instructed by the manager. In this way the network will form 'inside out', starting with at least one AP mote, then the one-hop motes, then two-hop motes, and so on. For any one mote to join, it need not ever be brought into range of the manager or AP motes, it only needs to be within range of any motes that are in the network.

Once joined, an ongoing *discovery* process runs continuously where motes listen for additional devices within range. During each discovery interval, a single mote transmits, and all others listen. Motes communicate this neighbor discovery information to the manager through a periodic *health report*, which gives the manager a stream of potential path information to use in optimization and network healing.

2.2.1 Mote ID

The 2-Byte mote ID is a nickname used in packets to avoid sending the full 8-byte MAC address over the air. The wireless network uses that nickname to save power and to maximize payload size for customer applications. The mote is given a mote ID by the Manager when it joins and the mote forgets it whenever it resets; the Manager reassigns an ID the next time it joins. The mote ID is NOT guaranteed to remain unique to that mote over its lifetime - only the MAC address uniquely identifies that mote.

In SmartMesh IP, mote IDs are handed out in the order motes join. The very first mote to join the network will always be ID 1 and will always be either the embedded manager or an AP Mote for the Vmanager. Subsequent mote IDs will be assigned in the order that devices join, be it regular motes or additional AP motes. The SmartMesh IP Manager does not preserve mote IDs through a Manager reset or power cycle - a mote will likely get a different mote ID every time the system is reset. While it may be convenient or less confusing to use the nickname to identify a mote for a person analyzing the network at a particular time (e.g. using the Command Line Interface or CLI), Manager API's use the 8-byte MAC address and a software application should always ALWAYS use MAC address for mote interaction.

2.3 Bandwidth and Latency

A SmartMesh IP network's total upstream data throughput is determined by how many packets per second can pass through the AP motes which act as the ingress/egress points in and out of the network. In the case of the SmartMesh IP Embedded Manager, which by definition contains a single AP mote, the maximum data throughput is 24 packets/s (without external SRAM) or 36 packets/s (with external SRAM). See the [Eterna Hardware Integration Guide](#) for SRAM details. The VManager supports any number of AP Motes, each of which supports up to 40 packets/s. For example, a four-AP VManager system can support up to 160 packets/s. This bandwidth is system wide and any mote can use any fraction of the total bandwidth. Once the total capability of the network is reached, the manager will deny any additional service requests. Each upstream packet supports a 90 B payload.

There are several settable parameters that influence how much upstream bandwidth (links) each mote will receive. These sometimes have complicated interactions. The system is designed to have the flexibility to both scale to large numbers of devices and to achieve ultra-low power for scavenging devices. In the case that the deployment has uniform data generation requirements at all motes, we recommend using the *basebw* parameter on the embedded manager (*basepkperiod* on the VManager) to set network-wide, uniform bandwidth allocation. When the requirements vary between motes, we recommend using the service model on each mote individually. Only for special use cases do we recommend changing the other values from their defaults. All devices are given a minimum of two upstream links regardless of any other constraints, and this includes the scenario where a mote has only a single parent.

In general, adding more links to motes:

- Decreases latency
- Increases packet/s throughput
- Increases power

There is no requirement to actually use all the bandwidth assigned to a mote. An application with low latency requirements can request more bandwidth than it needs to get additional links for itself and its ancestors and thereby decrease its upstream latency. Note that even if requested bandwidth is not used, it will count towards the total bandwidth available in the network. For example, 10 motes requesting 2.4 packets per second will completely fill a SmartMesh IP Embedded manager (without external SRAM), even if the motes actually only send in 1 packet per day. Because of the uncertainty in path stability, the network does not guarantee upper bounds on latency. Refer to the SmartMesh Power and Performance Estimator.xls to model networks and calculate expected latencies.

2.3.1 NumParents Parameter

The *numparents* parameter (default = 2) on the manager sets the number of desired parents for each mote. In optimizing the networks, the manager tests links to new parents for some motes, so occasionally motes will have more than *numparents* parents. In deployments where frequent environmental changes lead to high path failure rates, setting *numParents* to 3 or 4 reduces the chance of any mote disconnecting, but results in higher average mote power.



This parameter applies to the entire network and all motes will be connected to that number of parents. However, in a network with a single AP mote, there will always be one mote with only one parent. Similarly, if this parameter is set to 3, there will be one single-parent mote and one with two parents.

2.3.2 BWMult Parameter

The *bwmult* parameter (default = 300%) on the manager sets the provisioning safety margin in the network. This parameter applies to the entire network. If a mote is requesting to generate a packet every 12 seconds and *bwmult* is set to 300%, then the mote will be given at least one link per 4 seconds. Extreme caution should be used when lowering this below 300% as the vast majority of Dust deployments have been using this value for years. Conceptually, it allows motes to drop down to 33% path stability without filling up any packet queues. Lower settings should only be used for tight energy budgets requiring multi-hop networks.

2.3.3 BaseBW Parameter

The *basebw* parameter (default = 9000 ms) for the Embedded Manager or *basepkperiod* (default = 15000 ms) for the VManager is the interval between packets generated at all motes. This parameter applies to the entire network. The default setting is enough to carry all command and diagnostic packets as well as having each mote application generate one packet every 30 s. If a deployment has the same data generation rates at all motes, we recommend using *basebw* exclusively to set the network bandwidth.

2.3.4 Services

Applications wishing to support different data generation rates or run-time configurable data rates must use the services model. In this model, the mote API client (typically a microcontroller) is responsible for configuring the desired publish rate: the client could be preconfigured or an application message could be used to configure publishing dynamically - this is left to the OEM integrator. After opening a socket to the desired destination, the mote API client uses the *requestService* API to request a target packet interval and destination. The mote API client can expect a *serviceChanged* notification when the service level has been established or modified. In addition to respecting the granted bandwidth levels, it is expected that the sensor processor implement back-off to deal with API negative acknowledgments (NACKs) caused by network congestion.

The service may initially be granted at a level lower than that requested. In addition to the *serviceChanged* notification, the mote API client can use the *getServiceInfo* API at any time to query the status of pending request. The client could also notify a manager API client that the target service level has not been granted - however this kind of application level message is left to the OEM integrator to implement. The manager API client can also periodically query the *getMoteInfo* manager API to see the difference between requested and granted bandwidth - this information could be used to determine the cause of the service rejection, *e.g.* more repeaters are needed if existing 1-hop motes are link limited.

2.3.5 Cascading Links

All of the above parameters determine how many links a mote needs for its local traffic. If a mote has children, the network manager will add more links to carry the traffic of all its descendants. The calculation of this requirement is called *cascading* links. The manager looks at how many upstream RX links it has from its children, and adds the local traffic requirement to get the number of upstream TX links for that mote. In a multi-hop network the one-hop motes may have many more links than the leaf motes even if there isn't much traffic in the network.

2.3.6 Normal/Fast Downstream Modes

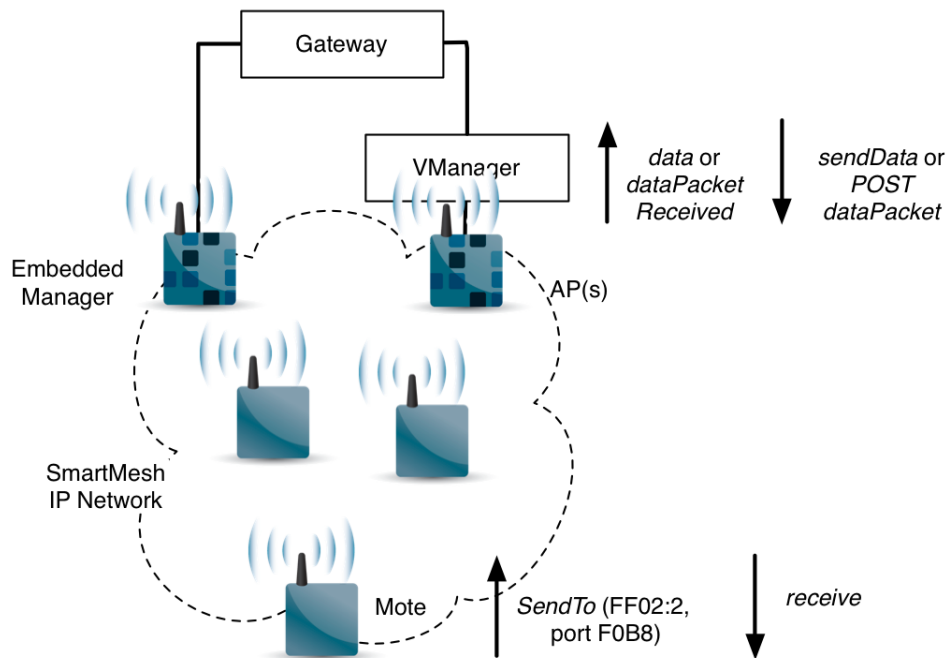
All SmartMesh networks require downstream control traffic during network formation. However, in networks used primarily for upstream sensor data collection, the downstream links are relatively unused after formation, and these unused links waste energy throughout the network at a rate inversely proportional to the downstream frame size. To reduce this waste, the SmartMesh IP manager supports an optional automatic transition from a "fast" downstream frame building phase, to a "normal" downstream frame operational phase. The default frame size for both upstream and downstream frames is 256 slots. The user can modify the *dnfr_mult* (*downframesize* in VManager) nonvolatile 'config' parameter via CLI (default = 1) to 2 or 4 to increase the normal size to 512 or 1024 slots respectively. This can also be done programmatically by calling the *setNetworkConfig* API with the *downFrameMultVal* parameter (or PUT /Network/Config/downFrameMultiplier in VManager). This lowers the average mote current by ~4 and 8 μA , respectively. As this feature, when enabled, limits the rate at which the manager can fix problems in the network, a setting > 1 should only be used in cases where the mote current using default settings is unacceptable.

2.4 Data Traffic

Motes use unreliable transport (UDP) for user data. However, this does not mean that data is likely to get lost. Downstream, messages sent to motes < 10 hops deep use source routing with multiple retries per hop. Packets sent to motes deeper than the source route limit are flooded after the source route is exhausted. In both cases, downstream packets have a small chance of not reaching the destination. In the case where this is unacceptable, application-layer acknowledgements can be used to ensure downstream delivery.

Upstream, packets are retried indefinitely at the link level until acknowledged. In rare cases (<0.001% typically) data that is accepted by a mote for forwarding is lost when a forwarding mote resets.

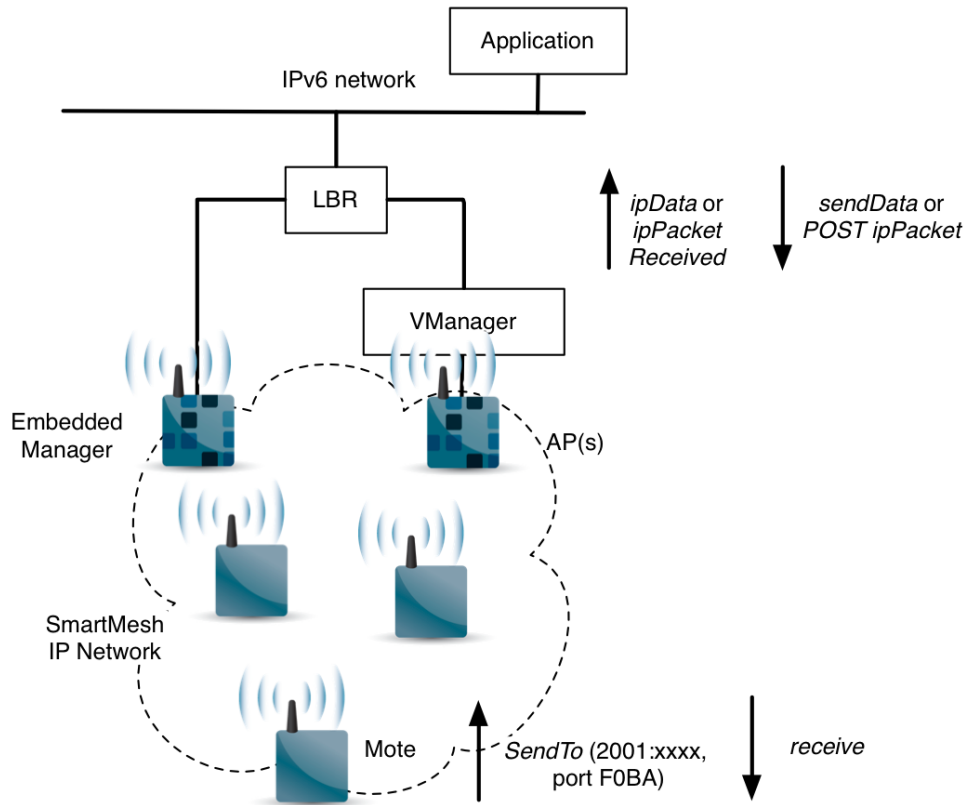
2.4.1 Directly Connected "Gateway" Application



An application connected directly to the manager can use an API to send packets to an arbitrary UDP port on a mote. With the *sendData* API (embedded manager), the application is only responsible for providing the destination, port, and payload. The destination address in this case is the EUI-64 of the mote, which is also the lower 8-bytes of its IPv6 address. When the API is called, a *callbackId* is returned - this *callbackId* is included in the *packetSent* notification (embedded manager) when this packet is injected into the network. A similar set of manager APIs (POST /motes/m/{mac}/dataPacket, GET /notifications *netPacketSent*) are found on the VManager. When the packet is received at the mote, a *receive* notification is generated which contains IPv6 source/port information in addition to the UDP payload. It does not contain the UDP checksum information as this is elided.

The sensor processor may use the *sendTo* API at the mote to send packets to the gateway via the manager by using the manager's well-known address (FF02::02) as the destination address. These result in a manager *data* notification (or VManager *dataPacketReceived*), which contains the EUI-64 address of the mote and port information, a timestamp, and the UDP payload.

2.4.2 End-to-End IP



A low-power border router (LBR, sometimes called an "edge router") connected to an IPv6 network is expected to generate the mesh and compressed 6LowPAN headers and to use the *sendIP* embedded manager API. The *sendIP* API also returns a *callbackId* when the packet is injected into the network. Again, there are equivalent VManager APIs (POST */motes/m/{mac}/ipPacket*, GET */notifications netPacketSent*). When the packet is received at the mote, a *receive* notification is generated which contains IPv6 source/port information in addition to the UDP payload. It does not contain the UDP checksum information as this is elided.

The sensor processor may use the *sendTo* API at the mote to send packets to an arbitrary destination. If the packet is addressed to any address other than the manager, the packet will result in an *IP data* notification (or VManager *ipPacketReceived*), which contains the EUI-64 address of the mote and a timestamp, but the 6LowPAN header is preserved in addition to the UDP payload.

2.5 Network Security

2.5.1 Security Layers

All packets in a SmartMesh network are authenticated on each layer, and encrypted end-to-end.

- Authentication - verifying that a message is from the stated sender, and that it has not been altered, or replayed
- Encryption - keeping payloads confidential

SmartMesh IP has several layers of security:

- Link-layer - packets are authenticated at each hop using a run-time key and a time-based counter - this ensures that only motes that are synchronized and accepted by the manager can send messages.
- End-to-end - packets are authenticated and encrypted end-to-end using run-time *session keys* and a shared counter - this ensures that only the intended recipient will understand the message (data privacy), and that replays, data corruption, or man-in-the-middle attacks can be avoided (data security).

When joining, motes send a *join request* to the network manager using a shared-secret *join key* known by the manager. Choice of keys is determined by the security mode, discussed below. The mote encrypts its join request with its join key, and the manager responds with a *session key* for end-to-end encryption of data traffic - this is known as the *security handshake*. Advertisements are a link-layer special case - they are authenticated with a well-known key to allow any new mote to authenticate them. Once a mote has joined with the correct join key, it receives four session keys that are used to encrypt network data in operation:

- A mote-specific session key used for network management traffic
- A mote-specific session key used for application traffic
- A broadcast session key used by all motes for network management
- A broadcast session key used by all motes for application traffic

Using these four keys, all regular data publishes are encrypted by the generating mote and can only be decrypted at the manager. Neither eavesdroppers nor routing motes can decrypt the packet data. Similarly, responses to manager commands can only be read by the manager. Downstream commands to a particular mote cannot be understood by routing motes or unintended recipients. Only commands sent explicitly to all motes in the network can be understood by all, being decrypted using the appropriate broadcast key.

2.5.2 Security Levels

SmartMesh IP has a choice of security levels that determine how the manager decrypts the join request. The manager can do this in three different ways:

- **Common Key:** The least strict security level is to accept a common join key. At this level, the manager will accept any mote that provides a join request encrypted with the common (network wide) join key. Dust Networks' network managers ship with a default common join key and Network ID that should be considered public knowledge. If the Network ID and common join key are left unchanged, overhearing and decrypting the packets that assign the session keys is difficult, however technically possible. Therefore, it is highly recommended to change the common join key to a secret one.
- **Access Control List (ACL):** The manager can also be set up to only accept motes on an access control list. The manager will take the join request, and first look for the serial number (MAC address) of the joining mote, and then decrypt the request with the associated join key. If both steps are successful, the mote will be accepted into the network. The ACL should be set up with a unique join key for every mote. This is the most secure method, but requires the most effort on the part of the commissioning workforce, since it requires that the manager and all the motes be configured prior to deployment in order to work together. If these devices are already configured correctly, the installer need take no action - the mote will join when it hears an advertisement. The ACL can also be set up with a common key - this provides some additional security over the common key alone, as each device's MAC address must be known to the manager for it to be able to join.
- **Common Key -> ACL:** Another strategy is to build out a newly installed network by allowing devices to join using a common key then switching to an ACL. This would be accomplished by constructing an ACL once the network is formed, assign a unique join key to each mote, and lastly change each mote's join key over the air using the *exchangeMoteJoinKey* command. This method is less secure during the initial installation, however much more secure during normal operation. Any additional motes added to the network later on would have to first be added to the ACL.

Password Management

Embedded Manager

The CLI interface requires a login, and the password entered determines the privilege used for the session. The default passwords match the two privilege levels: `viewer` and `user`. The `viewer` cannot make any configuration changes to the manager. The `user` has access to all commands. The login command can be used repeatedly without logging out to switch between privilege levels. Passwords for the two privilege levels can be changed using the `set config` command.

There is no login required for the serial API, as this is assumed to be a trusted connection.

VManager

User ID or username and password are used to limit access to all interfaces of the VManager, including the guest Linux system, the manager API and CLI. Many of these accounts have the default username and password `dust`, but the accounts are separate and need to be managed separately. The manager relies on two accounts - `dust` and `dustadmin`. You should presume that the default password for these accounts is public, and change them. The configuration database must be reloaded for these changes to take effect.

2.6 Blink Feature

2.6.1 Introduction



The Blink feature is available in SmartMesh IP Mote version 1.4.1. and later. It requires that the mote be programmed with loader 1.0.5.4 or later.

Blink is a SmartMesh IP mote feature allowing the application to publish packets through a standard SmartMesh IP network without joining. For the purpose of describing Blink, we will refer to two modes: Blink mode and Mesh mode, the latter describing the "normal" SmartMesh IP mode of operation.

Some of the key characteristics of the Blink feature:

- Lower average current than a standard Mesh mode mote, e.g. $\sim 2\mu\text{A}$ average for a Blink mode mote when sending one packet per day
- Blink data publishing must be kept infrequent, e.g. once per day or on an event
- Faster packet delivery from an un-joined state since no network joining process is required prior to sending data
- Same 99.999% data reliability as for data sent by Mesh mode motes joined to the network
- Security built in, using the Manager ACL security features
- Very large numbers of motes running in Blink mode can be deployed in a single network, up to 500,000

- Unidirectional upstream (i.e. data cannot be sent to a Blink mode mote)
- Use cases include basic monitoring, alarming, or using as a switch

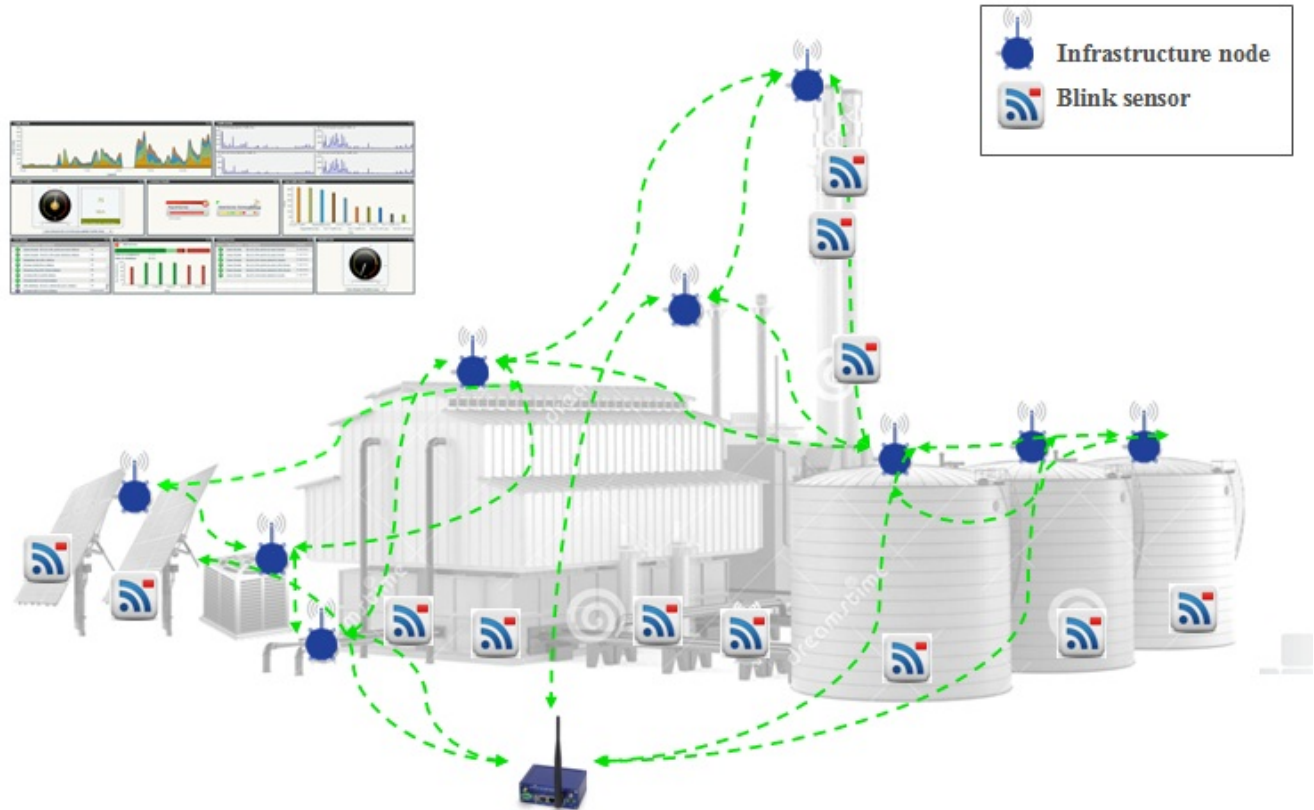
Blink is part of the SmartMesh IP mote software stack, which means that it is entirely up to the customer-developed application layer to select the operational mode. The application can decide to either join the network in Mesh mode and be part of the infrastructure or to operate in Blink mode and simply find a network through which to publish data, without joining it. The feature is accessible in **slave** mode via a UART API if an external μ Processor is used for the customer application, and as part of the On-Chip SDK where the mote is run in standalone mode with the customer application built in. An engineering demo version of the feature in **master** mode is also available and described later in this document. Blink mode motes do not join or participate in a network. They are simply stand-alone nodes that leverage a SmartMesh IP network within their range to send data.

2.6.2 When to use Blink

There are five key reasons for using the Blink feature as opposed to simply joining the network normally in Mesh mode:

1. Need for extremely low average current with low data rate devices, in this case less than $\sim 5\mu\text{A}$
2. Very large number of infrequently publishing devices otherwise beyond the possible network sizes manageable by a SmartMesh IP manager
3. Use as a switch to quickly send, e.g. an actuation signal, without the unnecessary delay of going through a full join process
4. Use as an alarm where no data is published in normal conditions
5. Use on mobile/roaming devices that would otherwise keep falling out of the network when they move

In all cases, a normal SmartMesh IP Mesh network infrastructure, consisting of motes running in Mesh mode, must be in place to accept Blink device data packets.



2.6.3 Detailed Description

When a SmartMesh IP mote is told to join a network with the *join* command, it searches for the network by listening for advertisements. Once a network with the appropriate NetworkID is discovered, the mote sends a join packet destined for the manager, requesting to join the network. What follows is a series of packets sent by the manager to establish a security session and establish a number of upstream and downstream links connecting this mote to other devices in the network. Once the handshake is complete, the mote is transitioned to the **Operational** state at which point it is allowed to both send and receive data.

When a mote is instead told to send a Blink packet with the *blink* command, it searches in exactly the same way as with *join*. However, once a single advertisement is heard, a Blink packet is sent instead of a join packet. The difference is the header identifying the packet as such, triggering the manager to push out the packet as a normal data notification, after verifying security credentials, instead of initiating a multi-step join process. If required, the customer application can send multiple Blink packets in a row without the need to repeat the search. A mote issued the *blink* command will be in the **Blink** state until it is either reset or issued a *join* command. Within the **Blink** state, the mote goes through three main phases: first it searches for an advertisement, second it sends the Blink packets provided by the application, and third it enters a low-power mode until receiving another *blink* command. The mote does not send keep alive packets during the second phase, so 60s after the last acknowledged MAC packet, the temporary links are cleared and the third phase begins, similar to a path failure event in a mote fully joined in a network. The *getTime* command can be called during either the second or third phases with the caveat that the mote clock is free-running since its last acknowledged MAC packet.

The following section details each key component.


Low data publishing frequency:

There are two key reasons for keeping Blink publishing frequency to a minimum:

- Power cost
- Shared bandwidth

The most energy consuming aspect of the Blink process is the search. The radio is on for a comparatively long time searching for advertisements from an existing nearby network. Once the mote hears an advertisement, the mote syncs itself to the device it heard and sends the Blink packet.

In a SmartMesh network, the communication between nodes is done through a set of scheduled links, some dedicated to pairwise communication either upstream or downstream, and some reserved for shared communication. Data publishing by a Mesh mode mote is done using these dedicated links which never interfere with other motes. Unlike normal data packets sent via dedicated links, Blink packets are sent using the join listen slots, which are provisioned as a shared resource. This use of a shared resource means that only so many motes can be trying to send a Blink packet at any given time. For example, having 10,000 Blink motes all trying to report their status at midnight would just not work.

 It is possible and very efficient to send multiple consecutive Blink packets since the search will only be done once. Be mindful that the packets are sent through a shared resource which means that any other nearby mote also trying to send Blink packets to the same network parent will contend for that same resource.

Lowest power operation:

The power cost of sending a packet is mostly incurred through the search process. As an example for a mote joining with 80% path stability and 3 potential parents, the mean search time is 18.3s, exponentially distributed. Using this average time, sending one packet once per 24 hours in Blink mode results in a mote average current of approximately 1.7 μ A. At a 12-hour interval the current goes to approximately 2.5 μ A, and at 1 hour it reaches 18.5 μ A. For a more detailed estimate of various use cases, refer to the "Blink Estimator.xls" Excel sheet.

99.999% data reliability:

The same mechanisms used for a Mesh mode mote are used in Blink mode. When a Blink packet is sent to the infrastructure node it hears, a link-level acknowledgment is returned. The packet then continues its journey through the wireless network, acknowledged at every hop, until it reaches the manager. Once the first link-level acknowledgement is received by the Blink mote, the probability of the Blink packet reaching the manager is greater than 99.999%.

Security:

In Mesh mode, unique session keys are assigned to each node. Because of Blink mode constraints, unique session keys are not assigned when used in this mode. The join key that is stored in the device is the only encryption key available. All packets are encrypted with this join key. It is highly recommended to assign unique join keys to each device manufactured rather than relying on a well known join key used by all devices.

An additional layer of security is required at the manager to prevent the participation of unauthorized devices. The ACL (Access Control List) security feature on the manager must be used, and the MAC address and join key for every device that is intended to take part in the network must be entered; this includes both devices operating in Blink mode and Mesh mode. If the MAC address of the device sending a Blink packet is not in the ACL, an error will be generated, and the packet will be dropped without generating a data notification. If the MAC address of a Mesh mode device is not in the ACL, the manager will not allow the device to join and will generate an error message.

Support for very large numbers:

Because these devices do not actually join and participate in the mesh network, the manager does not need to track or optimize the connectivity of these devices. As a result, Blink mode devices are not counted towards the maximum network sizes of the various manager configurations. See the Manager requirements section for details.

Upstream connectivity only:

Blink mode devices do not provide bidirectional data communication. While in Blink mode, they can only publish data, not receive it. If the need arises where a device wants to use two-way communication, the customer application at the mote can instead use the Mesh mode to join the network.

2.6.4 Manager Requirements

The embedded SmartMesh IP manager (with external SRAM) can support up to 1,200 Blink motes, and the large scale VManager can support up to 500,000. The embedded SmartMesh IP manager without external SRAM does not support Blink devices. The manager ACL must be populated with the MAC address and join key of every mote that is expected to participate in the network. Without an ACL entry for the intended Blink mode node, the Blink packets will not be delivered by the manager. Without an ACL entry for a mote joining the network using Mesh mode, the mote will not be allowed to join.



Manager versions that support Blink are:

- Embedded manager version 1.4.1 or greater.
- VManager version 1.0.1 or greater

3 The SmartMesh IP Mote

3.1 Introduction

SmartMesh IP Motes form the "body" of the network. The mote is responsible for:

- Maintaining synchronization to the network
- Forwarding data from descendants
- Generating health reports to continually update the manager's picture of the network
- Generating alarms to indicate failures, such as a lost path
- Presenting user interfaces to a sensor application

The [LTC5800-IPM](#) is a single-chip solution intended to be embedded in the customer's design. The [LTP5902-IPM](#) and [LTP5901-IPM](#) are modularly certified so they can be integrated without the need for radio certification. All motes interoperate with both the Embedded Manager and VManager.

3.1.1 Steps in a Mote Design

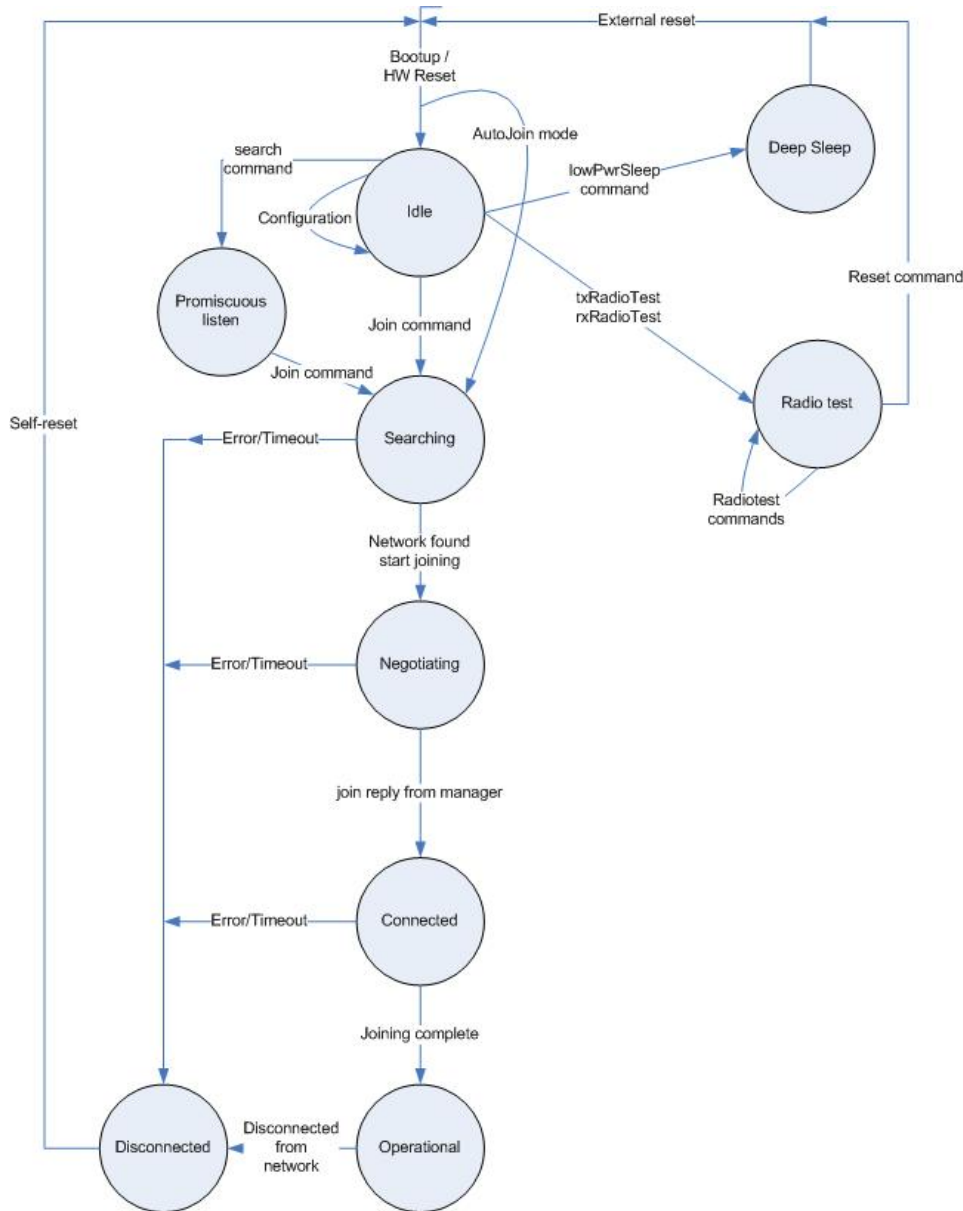
As with the manager, although the mote is busy with networking tasks, the customer's sensor application (either running on an external microcontroller, or running on-chip using the On-Chip SDK) has relatively few things it must do:

- Acknowledge the mote boot event
- Configure any parameters needed prior to join (such as *joindutycycle*)
- Use the *join* API to cause a mote to begin searching for a network
- Monitor the mote state to see when it is ready to accept data
- Request services in order to publish data
- Send data and respond to application layer messages

The [SmartMesh IP Mote API Guide](#) covers other commands to configure the mote. The [SmartMesh IP Mote CLI Guide](#) covers using the human interface to observe mote activity.

3.2 Mote State Machine

The following state machine describes the general behavior of a mote during its lifetime, and is provided for the user's information. In general an application only needs to issue the join command to enter a network, and issue a small subset of API commands to send data.



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 radiotest commands. It must be reset to return to normal operation.

3.3 Joining

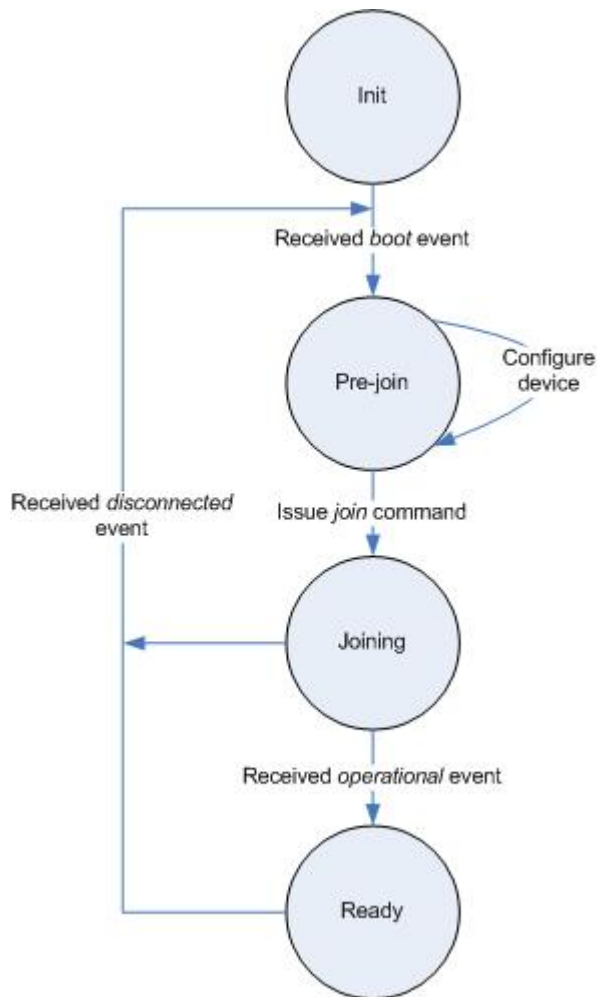
3.3.1 Programmatic Join

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 mote API 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, although this is not the lowest power configuration.
- **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 *setParameter* 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 then send data.

3.3.2 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, radiotest functionality cannot be exercised.

3.3.3 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 *setParameter<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 (*setParameter<networkId>*)
4. Start joining (*join* command)

3.4 Services

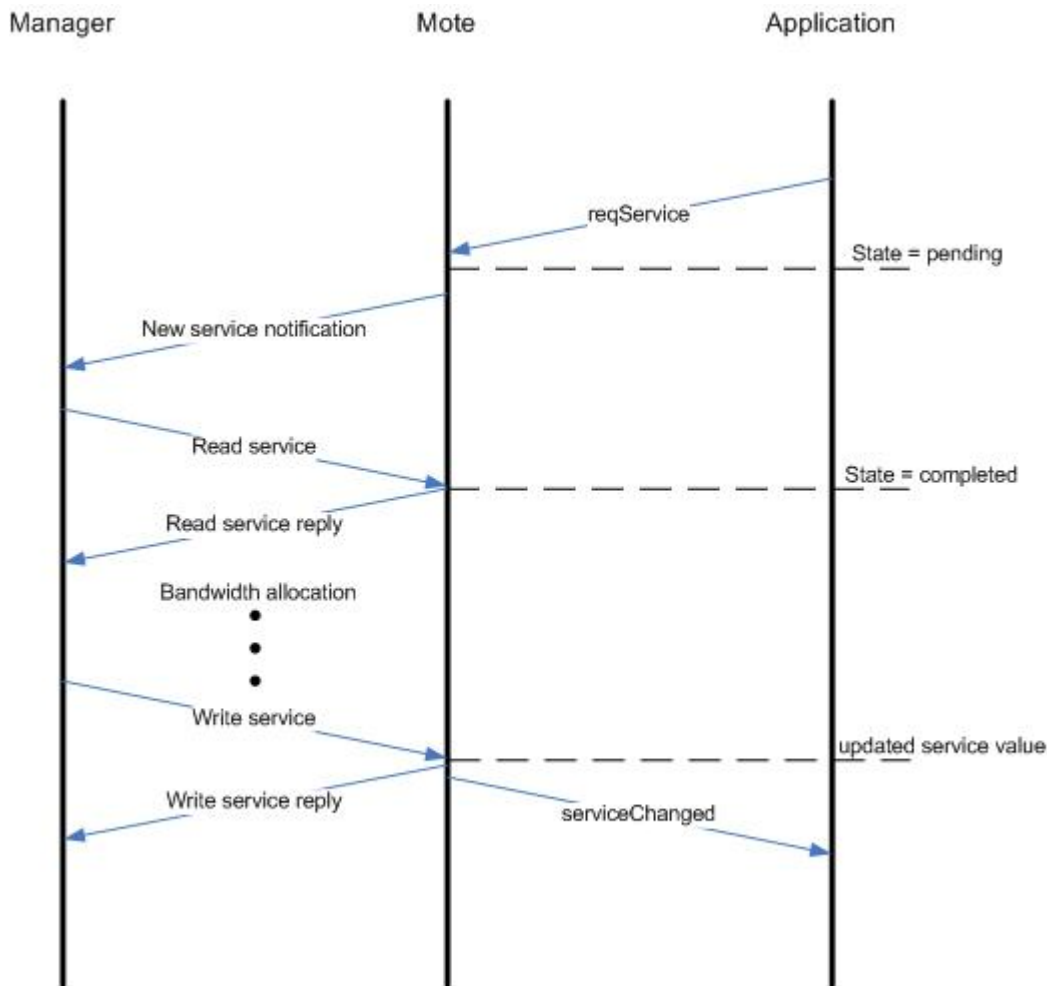
3.4.1 Requesting Bandwidth

Once a mote is in the network and has reached the **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**; this process can take up to a minute after all related bandwidth has been added to the network. 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:



3.4.2 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 application 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.

3.5 Communication

3.5.1 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* command to open a communication socket - this will give you a *socketID* for the socket. Currently only UDP sockets are supported.
2. Call *bindSocket* command to bind the socket to a port, the *destPort* you will use in the *sendTo* command.
3. Use *sendTo* command to send data. You will need to specify a *serviceType*, *priority*, and *packetId*, in addition to the payload and socket information. These are discussed in the *sendTo* documentation in the Mote API Guide. Currently only bandwidth (as opposed to latency) services are supported. Repeated calls to *sendTo* can be made on the open socket.
4. Call *closeSocket* command when you will no longer need to send data to that destination. This removes the port binding and frees any memory associated with the socket. It is not required to close a socket after each packet.

3.5.2 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 2-3 bytes of payload. The manager and mote must agree on one or more ports in order to be able to communicate - this can be arranged at runtime (using a well-known port such as 0xF0B8 to start) or be hardcoded into the application.

3.5.3 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 *ipData* 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 deliver all packets received on that port to the user's application using the *receive* notification.

3.6 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 the *setParameter<eventMask>* API call.

3.7 Factory Default Settings

The mote ships with the following factory defaults. The mote can be returned to factory settings by using the *restore* mote CLI command, or the *clearNV* mote API command.

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	0xFF (100%) for DC9003B, 0x40 (25%) for others
maxStCurrent	0xFFFF (no limit)
minLifetime	0 (no limit)
currentLimit	0 (none)
Auto Join	off

3.8 Power Considerations

3.8.1 Power Source Information

For full description, refer to the documentation of *setParameter<powerSrcInfo>* API.

Of the parameters described in the *powerSrcInfo* 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 μ A.

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 μ A of current throughout its lifetime but an on-board capacitor may be available to provide 100 μ A 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.

3.8.2 Routing Mode

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

3.8.3 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.

3.9 Master vs. Slave

3.9.1 Modes


Motes have two modes that control joining and command termination behavior:

- **Master**, in which the mote runs an application that terminates commands and controls joining. By default all the motes in [Starter kits](#) are configured for **master** mode. The API is disabled in **master** mode.
- **Slave**, in which the mote expect a serially connected device to terminate commands and control join - by default the mote does not join a network on its own. The API is enabled in **slave** mode, and the device expects a serially attached application such as APIExplorer to connect to it. If *autojoin* is enabled via *SetParameter* (SmartMesh IP only), a **slave** mote will join the network without requiring a serial application to issue a *join* command.

The mode can be set through the CLI `set` command, and persists through reset (*i.e.* it is non-volatile).

3.9.2 LEDs

For motes ([DC9003](#)) in **master** mode, the STATUS_0 LED will begin blinking immediately upon power up, as the mote will start searching automatically. When the mote has joined, STATUS_0 and STATUS_1 LEDs will both be illuminated. In **slave** mode, no LEDs may light - this should not be mistaken for a dead battery.

 LEDs of a [DC9003](#) board will only light if the LED_EN jumper is shorted. Master mode LED support available in SmartMesh WirelessHART mote version \geq 1.1.2.

3.9.3 Master Behavior

The **master** mode application can sample onboard analog and temperature sensors, read and write digital I/O, and contains a dummy packet generator containing a counter. By default, the application will sample and send temperature reports every 30 s. A customer application can interface with the **master** mode application via the OAP protocol, as described in the [SmartMesh IP Embedded Manager Tools Guide](#).

3.9.4 Switching To Slave Mode


By default, motes in starter kits ([DC9000](#) & [DC9021](#) and [DC9007](#)) and are configured for **master** mode. To read the current configuration, connect the mote to a computer via a USB cable and use the `get` mote CLI command. To configure the mote for **slave** mode, use the `set` mote CLI command:

Use the `get mode` command to see the current mode:

```
> get mode
master
```

Use the `set mode` command to switch to **slave** mode:

```
> set mode slave
> reset
```

 You must reset the mote for the mode change to take effect. Once set, the mode persists through reset.

3.9.5 Switching To Master Mode


To read the current configuration, connect the mote to a computer via a USB cable and use the `get mode` CLI command. To configure the mote for **master** mode, use the `set mode` CLI command.

Use the `get mode` command to see the current mode:

```
> get mode
slave
```

Use the `set mode` command to set the mote to **master** mode :

```
> set mode master
> reset
```

 You must reset the mote for the `set mode` command to take effect. Once set, the mode persists through reset.

3.10 Over the Air Programming (OTAP)

An external application can communicate with motes through the Manager and upgrade firmware on the devices – this is called Over-The-Air-Programming (OTAP). The SmartMesh SDK provides a reference application called *OTAP Communicator* that can be used to do OTAP with the Embedded Manager. Refer to the [OTAP Communicator documentation](#) for more details.

The basic steps of an OTAP operation are as follows:

1. Updated firmware is provided in the form of an `.otap2` file. See the [On-Chip SDK documentation](#) to build an `.otap2` file for customer generated code.
2. A handshake with all motes determines which motes can accept this update (the receive list), and prepares them for update.
3. The OTAP file is divided into packets and sent to all devices - only devices on the receive list will do anything with the file.
4. Process queries all motes on the receive list to verify that the image was received and is valid. Steps 2 and 3 are repeated until all (eligible) motes have received the file.
5. Sends the commit message to all motes that received the file. This will cause them to reprogram their flash with the new firmware.
6. Motes are reset and rejoin the network using the new firmware.

See "Building a SmartMesh IP OTAP Application" in the [SmartMesh IP Application Notes](#) for details needed to build your own OTAP application.

4 The SmartMesh IP Managers

The SmartMesh IP Manager is the "brains" of a SmartMesh IP network. The Manager is responsible for:

- Managing security information such as keys and nonce counters and distributing these to the network
- Determining the link schedule for every mote to ensure that time synchronization can be maintained and data service levels can be met
- Collecting health reports to continually update its picture of the network
- Responding to changes in topology
- Optimizing the network to minimize energy and spread traffic
- Presenting user interfaces to a network Host

There are two choices for a SmartMesh IP Manager:

- Embedded Manager
- VManager

The choices are distinguished by network size, bandwidth and the associated hardware. Both operate seamlessly with all SmartMesh IP motes and the functional and security aspects of all SmartMesh products are largely identical. The Embedded Manager and VManager are described in the following two sections.

The user chooses the Manager appropriate to their use case based on the following criteria:

Manager	Max Network Size	Access Point Motes	Data Throughput	Hardware/Software	Hot Failover	Can be synched to GPS time
Embedded Manager	100 motes	1 (onboard)	36 pkt/s	Runs on Eterna SOC	No	No
VManager	1000's of motes	multiple (remote)	40 pkt/s per APM	Runs on Linux VM	Supported	Supported

4.1 The SmartMesh IP Embedded Manager

4.1.1 Introduction

The SmartMesh IP Embedded Manager is a single-chip solution that incorporates management functions and the Access Point (AP) function on the same device. The modular LTP5902-IPR and LTP5901-IPR managers support networks of up to 32 motes, while the LTC5800-IPR can support networks of up to 32 motes by itself, or of up to 100 motes when external RAM is incorporated into the design. Use of external RAM also increases upstream data throughput by 30%, as more memory is available for link storage.

Manager	Pkt/s
LTC5800/LTP5901/LTP5902	24.3
LTC5800/LTP5901/LTP5902 + External RAM	36.1

Steps in Designing a Manager Client Application

A client application collects mote data and optionally monitors the health of the network and/or customizes the configuration of the network. Although the manager has many tasks it needs to do in order to manage a network, a client has relatively few. At a minimum, it should:


- Connect to the manager
- Configure any parameters needed prior to join (such as *networkID*)
- Subscribe to notifications to observe mote status and collect data

The [SmartMesh IP Embedded Manager API Guide](#) covers other commands to configure the manager, e.g. configure security (use of ACL), or collect detailed statistics from Health Report notifications. The [SmartMesh IP Embedded Manager CLI Guide](#) covers using the human interface to observe manager activity (including traces of mote state or data).

4.1.2 Accessing the Manager

Command Line Interface (CLI)

The SmartMesh IP Embedded Manager provides a command line interface that can be accessed through a terminal program (such as PuTTY). The CLI is intended for human interaction with the manager, *e.g.* during development to observe various traces. System parameters such as Network ID can be configured through the CLI. System and network status information can also be retrieved via this interface. CLI access is protected with user name/password login authentication that may be changed by the user.

 For more detailed information on connection to the CLI and the commands that are available, refer to the [SmartMesh IP Embedded Manager CLI Guide](#).

Application Programming Interface (API)

The SmartMesh IP Embedded Manager APIs provides a programmatic interface for interacting with the network. Host applications use the API to communicate with the manager. The embedded API port operates at 115200 baud, 8 data bits, no parity, 1 stop bit

Several tools are available for experimenting with and communicating with the manager API:

- The [SmartMesh SDK](#) provides Python tools for performing common tasks and experimentation with the API.
- The Serial API Multiplexer (Serial Mux) allows one or more clients to access the embedded Manager API through (possibly remote) TCP connections. Since the manager is accessed via a serial port, the Serial Mux allows for multiple clients to connect at the same time, e.g. a data logging application can be separate from a network health monitoring application and a network configuration application.
- The [Stargazer GUI](#) provides a graphical view of the wireless network and an interface for configuring periodic sensor data reports for the embedded manager.



For more detailed information on connection to the API and the commands that are available, refer to the [SmartMesh IP Embedded Manager API Guide](#).

4.1.3 Configuration and Usage

Network ID

Overlapping installations can be arranged into separate networks by assigning unique Network IDs to the motes and manager in each network. Motes won't communicate with motes or a manager which are using a different Network ID. The Network ID stored in the manager persistent configuration can be updated through the embedded manager API *setNetworkConfig* or the CLI `set config` command. The embedded manager must be restarted to use the new Network ID.

The embedded manager *exchangeNetworkId* command can be used to change the Network ID for all motes that are in a particular network. The *exchangeNetworkId* command reliably pushes a new Network ID to all the motes in a network. If a mote fails to acknowledge the exchange Network ID operation, the mote is removed from the network. The *exchangeNetworkId* operation may take several minutes to complete.

Network Time

The manager handles time synchronization in the network. The embedded manager automatically start the network when it boots, and network time will drift synchronously on all in-network devices compared to a host's system (which could be based on an "absolute" reference such as NTP or GPS) time.

There are two ways for an host application to query the current time on the embedded manager.

1. The application can trigger the TIMEn GPIO pin. When the manager detects the trigger, it sends a Time notification.
2. The application can call the *getTime* command. When the manager receives a *getTime* request, it sends a Time notification. Because of the delay of serial port I/O and command processing, this method is less accurate than triggering the TIMEn GPIO. The *getTime* API is provided for compatibility with systems that only communicate through the serial port API.

The networkTime notification contains the current embedded manager time in several formats:

- **ASN:** Absolute Slot Number, i.e. number of slots since ASN=0, which maps to 20:00:00 UTC July 2,2002.
- **uptime:** Manager uptime in seconds
- **Unix (UTC) time:** seconds and microseconds since Jan 1, 1970 in UTC

Communicating with Motes

The external network-side application is responsible for communicating with motes and with the attached sensor processors. The application uses the *sendData* embedded manager API to send packets to a mote and receives upstream data from motes by listening to data notifications.

Each packet in the network contains source and destination addresses and source and destination ports. Ports are used in the TCP/UDP sense of providing efficient filtering for messages destined for a particular service. Refer to UDP Port assignment section in the [SmartMesh IP User's Guide](#) to decide which ports to use for the application.

For a detailed example of using the upstream APIs, refer to the UpStream and SensorDataReceiver tools in the SmartMesh SDK.

4.1.4 Network Activity

Network Structure and Formation

The SmartMesh IP Manager performs automatic network management operations to maintain the health of the network as well as optimize it for the lowest power consumption and highest reliability. The manager also dynamically makes changes to the mesh as conditions in the network vary, such as path stability changes due to interference, addition and removal of service requirements, and addition and removal of devices. Customer applications never have to get involved in the management aspects of the network.

If motes are powered up in the vicinity of a manager, they will start joining almost immediately. The AP advertises on average twice per second and motes, once joined, advertise once every two seconds. All motes will advertise except for those explicitly designated "non-routing". The manager does not deactivate advertising; if desired, advertising must be deactivated by the application and can save power on each mote. Such an application must be able to detect lost motes and reactivate advertisement to retrieve them.

In order for a mote to join a manager's network, the mote must be assigned the Network ID used by the manager. The manager and mote exchange several messages in the join process to establish session security keys and proper network routing. The mote goes through several state changes during the join process, ending in the **Operational** state. The Manager will detect if a mote leaves the network and will update the mote's state to **Lost**.

An external host application can keep track of motes joining and leaving the network by listening to *Event* notifications.

Network Health

The motes and manager each keep track of network and device statistics. The motes generate *Health Reports* that are sent to the manager, who aggregates the resulting statistics on a per-mote and network wide basis.

Statistics queries are available through the following embedded manager API commands:

- *getNetworkInfo* provides network-wide statistics.
- *getMoteInfo* provides statistics accumulated from communicating with a mote.
- *getManagerStatistics* provides API connection statistics.

From the CLI, the `show stat` embedded manager command (shown below) summarizes all of these commands.

```

> show stat
Manager Statistics -----
established connections: 1
dropped connections    : 0
transmit OK           : 7971
transmit error        : 0
transmit repeat       : 0
receive OK            : 40
receive error         : 0
acknowledge delay avrg : 19 msec
acknowledge delay max  : 112 msec
Network Statistics -----
reliability: 100% (Arrived/Lost: 7998/0)
stability: 95% (Transmit/Fails: 5663/290)
latency: 1000 msec
Motes Statistics -----
Mote   Received   Lost  Reliability Latency Hops
#2      1726        0    100%       990   1.0
#3      3143        0    100%      1190   1.0
#4      3020        0    100%       800   1.0

```

Network Statistics

- *reliability*: The network reliability is calculated as the percentage of packets generated by any mote that are successfully received by the manager. This number should be 99.99% or higher in a healthy network. This statistic is counted directly on the manager by monitoring the security counter on the packets as they come in.
- *stability*: The network path stability represents the percentage of total MAC transmissions that have succeeded. This number will vary depending on individual mote placement. The network is designed to achieve 100% reliability even at 50% stability. At lower stability values, motes will use more energy and bandwidth as more packets need to be retried. This statistic is computed based on health reports received from motes. The manager increments the total number of transmits and fails with each new health report arrival.
- *latency*: The average upstream latency of packets received by the manager. The manager checks the packet header for the generation timestamp (ASN) and compares this to the current ASN to calculate each individual packet latency.

Mote Statistics:

- The mote statistics show a breakdown of each of the network statistics in a mote-specific manner. Additional to the statistics shown for the entire network, the manager also print the average number of hops that an upstream packet takes from mote to manager. This is calculated by looking at the TTL field in the Mesh header which is decremented at each wireless hop.

The manager's statistics can be reset using the *clearStatistics* API or the CLI `exec clearStat` command.

Health Reports

Motes periodically report statistics to the manager in health reports. The manager uses the health reports to determine a mote's neighbors and path stability for network optimization. The manager accumulates some lifetime statistics but the manager does not store individual health reports.

Health reports are available as notifications in the API, so an external application connected to the manager can subscribe to health report notifications and track network statistics in detail over time.

Optimization

Based on information in the health reports, the manager continuously works on improving the network. For each mote, the manager considers how the existing parents compare to other neighbors discovered by the mote. If one existing parent looks better than the other, or if a discovered but unused neighbor looks like it would make a better parent, the manager will add a link from the mote to the better parent. This is called the *optimization add* cycle. Comparing the parents involves comparing a scoring function that attempts to minimize mote energy consumption while spreading traffic across the busiest motes. During the add cycle, the manager often has to guess at which paths are going to be best for the network and tries them out by adding single links to each mote.

After an hour, the manager has collected four health reports from the mote with the new parent and then has sufficient data to quantify which parents are best. Any mote with an extra link at this stage will delete the link to its worst parent, again by the same scoring metric. This is called the *optimization delete* cycle. The process continues for the life of the network, alternating add-delete-add-delete, each one hour apart. Running continuously allows the network to adapt to any changes in the environment.

4.1.5 Access Control


ACL Management

Motes use a symmetric join key to encrypt the initial join request when joining a network. If the join key used by the mote does not match the mote's join key configured on the manager, the manager will not be able to decrypt the join message and will not allow the mote to join the network. By default, motes and the manager ship with a common join key set to a default value - see the [The SmartMesh IP Network](#) section of this guide for more details on security.

The manager maintains an Access Control List (ACL) which associates a mote's MAC address with a mote-specific join key. Once an ACL entry has been set, the manager will not let motes join the network that are not in the ACL. The join key for a mote in the network can be updated on both the manager and mote using the embedded manager *exchangeMoteJoinKey* command. The manager pushes a new join key to a mote, and if the mote doesn't acknowledge the update, then the manager will not update its ACL. A mote can be added manually to the ACL through the *setACLEntry* command in the API or the *set acl* command in the CLI. See the [related documents](#) section of this guide for links to the manager and mote API and CLI guides.

Aside from pre-configuring the motes and manager as a set, or using out-of-band tools to construct an ACL, the most straightforward procedure for setting up ACL for all the motes in a network is with the following steps:

- Let motes join the network using the common join key.
- For each mote in the network, use the `exchangeMoteJoinKey` command to set an ACL entry for each mote in the network.
- Once all the motes have successfully completed the exchange join key operation and have an entry in the ACL, reset the manager and ensure the motes join.

 The maximum number of entries in the ACL is 1,200


4.1.6 Restoring Manager Factory Default Settings

The manager can be restored to factory settings using the `exec restore` CLI command or the `restoreFactoryDefaults` API command. The following table lists defaults for the CONFIG parameters affected by restoring to factory settings. Restoring also clears the access control list.

Parameter	Comment	Default
netid	Network ID	1229
txpower	AP Tx Power [dBm]	8
frprofile	Frame Profile ID	1
maxmotes	Max number motes (includes AP)	33 (100-mote managers will revert to this state, and need to have maxmotes manually set back to 101)
basebw	Base bandwidth - period between packets [msec]	9000
dnfr_mult	Downstream Frame Multiplier	1
numparents	Minimum number of parents	2
cca	CCA value	0
channellist	Channels list	7FFF (hex)
autostart	Autostart On/Off	1 (on)
locmode	Reserved	0
bbmode	Backbone mode	0 (off)
bbsize	Backbone frame size	1

pwdviewer*	Password of viewer user	viewer
pwduser*	Password of regular user	user
commjoinkey*	Common Join Key	44 55 53 54 4E 45 54 57 4F 52 4B 53 52 4F 43 4B (hex)
ip6prefix	IP6 address	::
ip6mask	IP6 mask	FFFF:FFFF:FFFF:FFFF:: (hex)
radiotest	Radio test	0 (off)
bwmult	Bandwidth provisioning multiplier [%]	300
onechannel	One Channel Network	FF (hex)

4.1.7 Channel Blacklisting

 Although the network may operate on as few as five channels, it is recommended that the network run on as many channels as possible for greater resiliency and more overall bandwidth. Starting with embedded manager version 1.2.2, the manager requires that at least 7 channels remain after blacklisting.

The default behavior for SmartMesh Networks is to blacklist only the sixteenth channel (2480 MHz) to comply with requirements in the United States (as regulated by FCC) and Canada (as regulated by IC). The *setNetworkConfig* embedded manager command is used to specify a bitmap of which channels are used. You are responsible for ensuring that the allowed frequencies conform with local RF regulations.

Channel 0 (2405 MHz) corresponds to IEEE channel 11, and Channel 15 (2480 MHz) corresponds to IEEE channel 26.

4.2 The SmartMesh IP VManager

4.2.1 Introduction

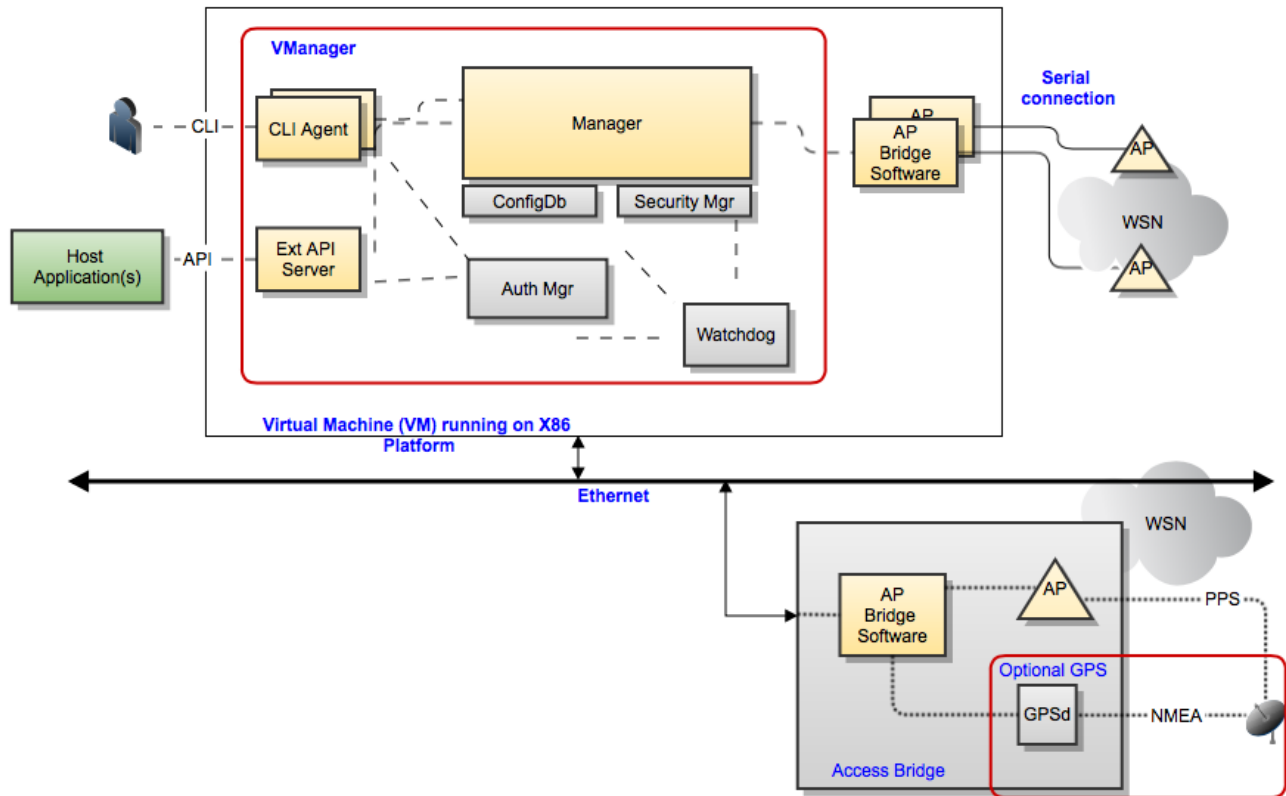
While the SmartMesh IP Embedded Manager form factor is convenient for many applications, it has some limitations due to resource constraints associated with being a small SoC. The VManager is a hardware-independent software manager that resolves the limitations of a single chip solution. The VManager runs on a range of Linux-based hardware platforms. The feature highlights of the VManager are:

- Large scale networks – thousands of motes
 - Achieved by using multiple AP motes
- High throughput networks
 - Achieved by using multiple AP motes, 40 pkt/s each
 - Significant downstream bandwidth increases with new network configuration options
- Hot Redundancy
 - AP mote redundancy by simply co-locating pairs
 - Manager redundancy with multiple instances (Future feature)
- Support for discontinuous network segments
 - Shared sense of time across all network segments by using optional GPS synchronization

High Level Architecture

The VManager software comes built as a complete software system within a virtual machine. Virtualizing the environment simplifies installation and platform compatibility issues. Both VMware and Oracle's Virtual Box Virtual Machine hosts are currently supported.

The manager module, the CLI interface, and the host application API interface all run inside a Virtual Machine on an x86 based hardware platform. The AP mote functionality is on the LTx5xxx parts acting as the bridge to the SmartMesh IP wireless mesh. A new software agent called AP Bridge Software now logically connects the VManager and the AP functionality.



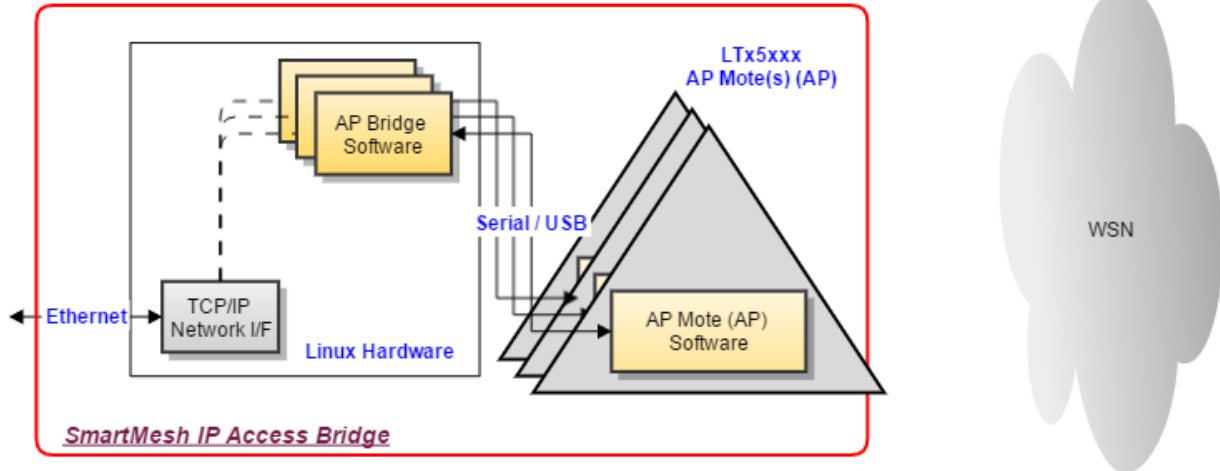
High Level Architecture

The VManager provides great flexibility in connecting to AP motes. AP motes can be installed on the same machine running the VManager using serial or USB ports, or can be installed anywhere on the internet when running on a separate small computer. Additionally, any number of AP motes can be installed to a single VManager instance depending on the application and overall system requirements.

The Access Bridge

AP Motes can be installed either directly on the same platform running the VManager, or on entirely separate TCP/IP connected Linux platforms. When installed locally, the AP Bridge Software already installed on the VM can be configured for any number of devices.

When installed separately, then the external platform referred to as a SmartMesh IP Access Bridge must be included in the customer's design. A representative architecture is shown below:



Access Bridge Architecture

The blocks in yellow represent software that is provided from Linear. The AP Mote Software is available as a binary and can be programmed onto any LTC58xx or LTP59xx device. The AP Bridge Software is available as C source code, ready to be integrated and built into any customer platform.

As an example, a fully functional Access Bridge has been built on a Raspberry Pi 2 and is available as a single binary. This is a way for a customer to start testing a system with external AP Bridges.

4.2.2 Access Point Motes

The VManager is able to manage networks containing multiple Access Point (AP) Motes. Each network requires at least one AP Mote. Adding AP Motes improves network performance , including:

- Higher throughput (pkt/s) upstream and downstream: 40 pkt/s of network capacity per AP Mote
- More motes in the network: the number of motes per AP Mote varies depending on various parameters, including publish rates, number of devices in the network, and network topology; refer to the [SmartMesh Power and Performance Estimator](#) spreadsheet for details
- Lower latency upstream and downstream: spreading AP motes across a large area reduce networks hop depth
- Precise UTC timing at the motes: UTC time can be available at each node in the network if an optional GPS is added to an AP Mote
- Redundancy: other AP Mote(s) will seamlessly take over if any AP Mote fails

Access Point Mote Timing Configurations

The simplest way to build a network with multiple AP Motes is to leave their Clock Source set the default Auto value (clkSrc=3). This configuration satisfies most application scenarios.

The alternative Clock Source is GPS (clkSrc=2) which should only be used if the application needs to do one of the following:

- Synch every mote in the network accurately to UTC time, e.g. for synched time stamping or data acquisition
- Build a single network where sub-sections are physically separated enough to be outside of radio range of the others, e.g. sub-networks in different buildings or even in different cities

If the VManager is configured in GPS mode, at least one AP Mote must be configured with its Clock Source set to "GPS" (clkSrc=2) and have an attached GPS module to provide a PPS (pulse per second) signal and UTC time. All other AP Motes in the system can either be set as GPS or Auto. Note that regardless of the mode chosen, timing between nodes within the network will be synched precisely, per the SmartMesh IP product datasheets.



If any GPS-enabled AP Motes are used in the system, the VManager must be configured in GPS mode using the `config` command as follows;

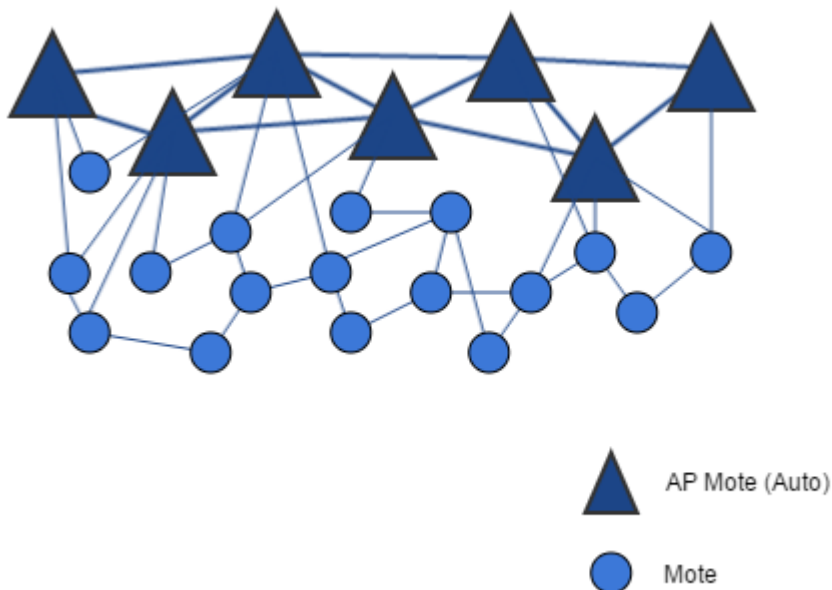
```
$> config set network gpsMode=True
```

When building physically separated sub-networks that are managed by the same VManager, use at least one GPS enabled AP Mote per sub-network. As an example, a network consisting of 100 motes in one building and another 100 in a completely separate building would require at least one GPS enabled AP Mote per building.

- ⚠ 1) AP Motes configured as Auto are only permitted to join other AP Motes as parents, so while multiple hops can be spanned by a mesh of AP Motes, they cannot be placed far enough away from each other such that they would create different sub-networks
- 2) GPS-enabled AP Motes will join the network faster since they have an absolute sense of time and do not need to hear another AP Mote to join

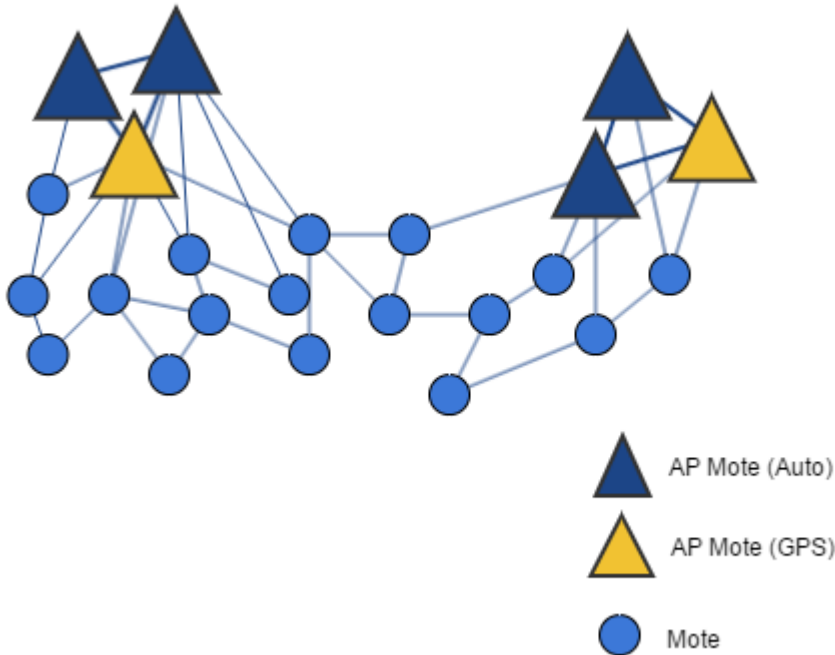
Deployment Rules of Thumb

The minimum number of AP motes required is determined by the traffic requirements for the network. Additional AP Motes can be deployed to create redundancy or to reduce latency in a network by lowering hops to an egress point. If a primary goal for the network is to ensure reliability, and the network does not have more than 2-3 hops, then it is recommended to co-locate all AP Motes with their antennas roughly 1m apart from each other. If a primary goal for a multi-hop network is to reduce latency, then place AP Motes throughout the environment spanned by the network ensuring that they are close enough to other AP Motes to form a connected mesh. Referring to the example below, if there are 7 Auto AP Motes available, they should be placed in order to form an AP mesh.



Network with Mesh of AP Motes

Placing them in separated clusters requires GPS-enabled AP Motes as this will form two sub-nets. For example, they can be placed as in the following figure where the gold-colored AP Motes are GPS-enabled, to form two sub-nets which can be managed by a single VManager. Note that only GPS-enabled AP Motes can be deployed without a direct connection to other AP Motes.



Network with Two AP Mote Subnets

4.2.3 Installation

The VManager is provided as a complete virtual machine, which includes the Linux operating system, the VManager, Access Bridge software, Swagger API documentation, and all other required tools. Installation involves first installing a virtual machine host and then importing the VManager image. Both the VMware and Oracle VirtualBox platforms are currently supported.

VManager Download

The VManager can be found in two separate files as follows:

- VManager_xxxx.ova: This is a full Virtual Machine image containing the VManager
- SmartMesh IP.zip: This file contains all IC based software that is required for the AP

Both files are available for download through your [MyLinear](#) account. Contact your local sales representative to gain access through your myLinear document locker.

VManager Installation - VirtualBox

1. Download **VirtualBox version >= 5.x** at <https://www.virtualbox.org/wiki/Downloads>
2. Download the **VirtualBox Extension Pack** from the same site

3. Install both VirtualBox and the Extension Pack on your machine as instructed by the installation software, a reboot will be required
4. Download the VManager package from the myLinear document locker as described above
5. Import the VManager instance into VirtualBox
 - Under **File** --> **Import Appliance...** select the file `VManager_XXXX.ova`
 - Make NO changes to the settings and click the **Import** button
6. **Start** the virtual machine now installed called "VManager"

VManager Installation - VMware

1. Download VMware Workstation Player for Windows at https://my.vmware.com/en/web/vmware/free#desktop_end_user_computing/vmware_workstation_player/12_0
 - Note: It is possible to run as a free player, but a license should be purchased for commercial use
2. Install VMware Workstation Player on your machine as instructed by the installation software
3. Download the VManager package from the myLinear document locker as described above
4. Open the VManager instance from VMware Workstation Player
 - Under Player -> File -> Open, select the file `VManager_XXXX.ova`
 - Click the Import button
 - If you see a pop warning "*The import failed because ... did not pass OVF Click Retry to relax OVF ...*"
 - click Retry
5. Select the virtual machine
 - Click "*Edit virtual machine settings*"
 - Under the "Hardware" tab -> "Network Adapter", Select "*Bridged: Connected directly to the physical network*"
 - Click OK
6. Start the virtual machine called "VManager" by selecting it and clicking the "Play" button

AP Mote Programming & Configuration

If you are using the latest DC9021B evaluation kit, it includes a [DC2274A-B](#) which is already programmed and configured as an AP Mote. This step can be skipped.

If you are using a [DC2274A-A](#) reference board or customer built AP Mote board can also be used. A standard DC90xx reference device cannot be used since it is missing some connections.

The AP Mote hardware is the LTC58xx IC, LTM58xx micromodule, or LTP59xx module programmed with the AP software binary. Unlike the DC90xx reference devices that are available in the SmartMesh IP kits, the AP Mote PCB requires that two additional signals be made available to the AP Bridge software process, namely the TIMEn pin for a PPS (Pulse-Per-Second) from an optional GPS device, and the reset pin. The [DC2274A-A](#) and [DC2274A-B](#) reference boards contain these additions and support the AP Bridge software.

Programming a [DC2274A-A](#) as an AP Mote:

- Install the Eterna Serial Programmer Utility (ESP). It is available in the `SmartMesh Tools.zip` file in your MyLinear document locker. You can extract its contents anywhere convenient.

- Connect your [DC2274A-A](#) to a USB port - if this is the first connection, Windows should launch the FTDI driver installer. If not, see the troubleshooting section of the SmartMesh IP VManager User's Guide.
- Copy the `prog_APM_DC2274A-A.bat` file and supporting image components from the SmartMesh IP.zip file (`../Eterna/AP Mote`) into your ESP folder.
- Run the `.bat` file to re-program your [DC2274A-A](#) into an Access Point Mote.

For default operation without using GPS as a time source, the AP Mote can be used "as-is" in its default state. If however the network is configured with a GPS time source, the AP Mote(s) will require configuration, the clock sources must be assigned as described in the "Access Points" section of the SmartMesh IP VManager User's Guide. The configuration instructions for the AP Mote are in the "Configuring the Gateway" section of the "VManager AP Bridge User's Guide".

AP Mote Installation - On Local Host

The AP Bridge software is already installed on the VManager VM and will automatically be used if an Access Point (like a [DC2274A-B](#)) is connected directly to that host computer. If an AP Mote is to be used locally, i.e. installed on the same computer as the VManager instance, then follow these steps:

- Enable port forwarding in the virtual machine. The VManager virtual machine must not be running.
 - In VirtualBox --> go into the **Settings** menu window and select **USB**, then enable the USB ports (select USB 3.0)
 - In VMware --> go into "Edit virtual machine settings" and select "USB Controller" under the Hardware tab, choose USB 3.0 for USB compatibility.
- Plug one or more AP Mote(s), such as the [DC2274A-B](#), into a USB port
- Start the virtual machine
- Attach the USB [DC2274A](#) AP Mote(s) to the VM
 - In VirtualBox --> In the **Settings->USB** menu, click the "+" sign on the right and select the the evaluation board(s) that appear
 - For example --> LTC DC2274A WITH MEMORY 60xxx [0800]
 - In VMware --> From the virtual machine screen's top right corner, right click the USB stick icon, select "*Connect (Disconnect from host)*"
- Open an SSH window to the VManager machine with Putty (or similar application)
 - In VirtualBox --> Connect to localhost:2222 (from the host running the VM)
 - In VMware --> Connect to <Host IP address>:22
- Login using these credentials
 - User = `dust`
 - Password = `dust`
- Execute the following commands to configure the AP Mote(s) installed by typing the input after the prompt (your \$ prompt line may differ). Note that the following shows an example where two AP Motes are connected to the system.

```

$ update-apc-config auto
  --> You will be prompted for the dust user password ... "dust"
installed udev rules, APC configuration will be created automatically when an AP mote is connected
Create APC configuration for LTC_DC2274A_WITHOUT_MEMORY_603528
  supervisor conf file created
  APC conf file for apc-603528 created
apc-603528: available
apc-603528: added process group
Create APC configuration for LTC_DC2274A_WITHOUT_MEMORY_603983
  supervisor conf file created
  APC conf file for apc-603983 created
apc-603983: available
apc-603983: added process group
No config updates to processes


```

- To verify that each AP Bridge instance (APC) has been correctly installed, run `update-apc-config` which should yield the following results:

```

$ update-apc-config
Name           Host           Port    api-device    reset-device
-----
apc-603528    localhost     9100    /dev/serial/by+ /dev/serial/by+
apc-603983    localhost     9100    /dev/serial/by+ /dev/serial/by+

```

 The AP Bridge software is configured by default to use the *Auto* clock source for all AP Motes in the system. No changes are needed unless GPS timing is used at the AP Motes.

AP Mote Installation - with an external AP Gateway

An AP Mote can be connected to the VManager through a SmartMesh IP Gateway. A fully functional reference Gateway system has been built for the Raspberry Pi. Additionally, the entire project including source code for the AP Bridge Software is provided so that a Gateway can be built with any number of other systems.

Refer to the "VManager AP Bridge User's Guide" for details of the installation and configuration process.

Verifying the VManager Installation

The following steps can be used to verify that the VManager is running and AP Motes have joined the network.

Step #1

Open an SSH window to the VManager system. Depending on how the networking for the VManager VM is configured, you may be able to SSH directly to the VM's IP address, or you may need to connect to port 2222 if the VM is sharing the host's network connection.

```
$ ssh dust@localhost -p 2222
or
$ ssh dust@192.168.1.10
```

- Use the `smctl status` command to verify that all of the necessary processes are running:

```
$ smctl status
apc-1                RUNNING      pid 1015, uptime 0:03:11  <-- Will only appear when a
local AP Mote is installed
apiserver            RUNNING      pid 1210, uptime 0:02:39
authmanager          RUNNING      pid 1155, uptime 0:03:09
configdb             RUNNING      pid 1134, uptime 0:03:10
manager              RUNNING      pid 1172, uptime 0:03:07
watchdog             RUNNING      pid 1012, uptime 0:03:11
```

- If any of these processes are NOT listed as "RUNNING", restart them using `smctl restart`. It will take about a minute for all the processes to restart.

Step #2

Open a Manager CLI client called **console** using the same SSH window (or open a new one as described in the steps above):

```
$ console
Welcome to the Voyager CLI Console on Linux
Version 1.0.0.1

Enter your username:
```

Login to console using the following credentials:

- Username = dust
- Password = dust

You are now logged into the manager CLI. As a simple test, run the `sm` command to see if any motes have joined the network. If so, the command will display a table. In the following example, 3 AP Motes are joined to this manager, one set to Internal clock (master time keeper) and the other two as Network Clock (time followers).


```

$> sm
AP MAC                               Id Clk State  State time  Age Jn  Nbrs Links
-----
00-17-0D-00-00-60-35-28             1 Int  Oper   0-18:14:46  NA  1   18   57
00-17-0D-00-00-60-37-78             8 Net  Oper   0-18:14:38  NA  1   18   41
00-17-0D-00-00-60-3C-4C            19 Net  Oper   0-18:14:20  NA  1   18   40

Mote MAC                              Id State  State time  Age Jn  Nbrs Links
-----
00-17-0D-00-00-38-0F-CC              2 Oper   0-00:03:00  NA  3    3   12
00-17-0D-00-00-60-09-D5              3 Oper   0-18:13:03  NA  2    3   12
00-17-0D-00-00-60-06-2B              4 Oper   0-18:13:23  NA  2    3   12
00-17-0D-00-00-60-09-85              5 Oper   0-18:13:12  NA  2    3   12
00-17-0D-00-00-60-03-4B              6 Oper   0-18:13:21  NA  2    3   12
00-17-0D-00-00-60-04-CB              7 Oper   0-18:13:05  NA  2    3   12
00-17-0D-00-00-60-04-B2              9 Oper   0-18:13:29  NA  2    3   12
00-17-0D-00-00-60-04-B0             10 Oper   0-18:13:06  NA  2    3   12
00-17-0D-00-00-60-06-24             11 Oper   0-18:13:25  NA  2    3   12
00-17-0D-00-00-60-03-A5             12 Oper   0-18:13:09  NA  2    3   12
00-17-0D-00-00-60-05-E2             13 Oper   0-18:13:34  NA  2    3   12
00-17-0D-00-00-60-04-80             14 Oper   0-18:13:08  NA  2    3   12
00-17-0D-00-00-60-09-83             15 Oper   0-18:13:03  NA  2    3   12
00-17-0D-00-00-60-02-F1             16 Oper   0-18:13:14  NA  2    3   12
00-17-0D-00-00-60-04-CF             17 Oper   0-18:13:25  NA  2    3   12
00-17-0D-00-00-60-06-F1             18 Oper   0-18:13:28  NA  2    3   12

APs: 3, Motes: 16, 19 live, 0 joining

```

 The VManager's networkID is set to 1229 by default, which is the same default as the Embedded SmartMesh IP Manager, and motes. Refer to the [SmartMesh IP VManager CLI Guide](#) to change the networkID to a different value.

Troubleshooting

Windows FTDI Driver installation

Devices communicate with your computer using a serial connection via USB. When you connect the device to your computer, you should be asked to install a driver for it. Because the device uses a serial chipset from Future Technology Devices International (FTDI) which is found in many different devices, it is possible that you already have a version of the FTDI drivers installed on your machine.

If you don't have the drivers installed, download the version appropriate for your operating system from <http://www.ftdichip.com/Drivers/VCP.htm>. We recommend you download the driver files on your computer's desktop.

- ✔ Once you have installed the driver, use the same USB port each time you reconnect the device to the computer. If you connect to a different USB port, you will need to repeat the following procedure for that port.

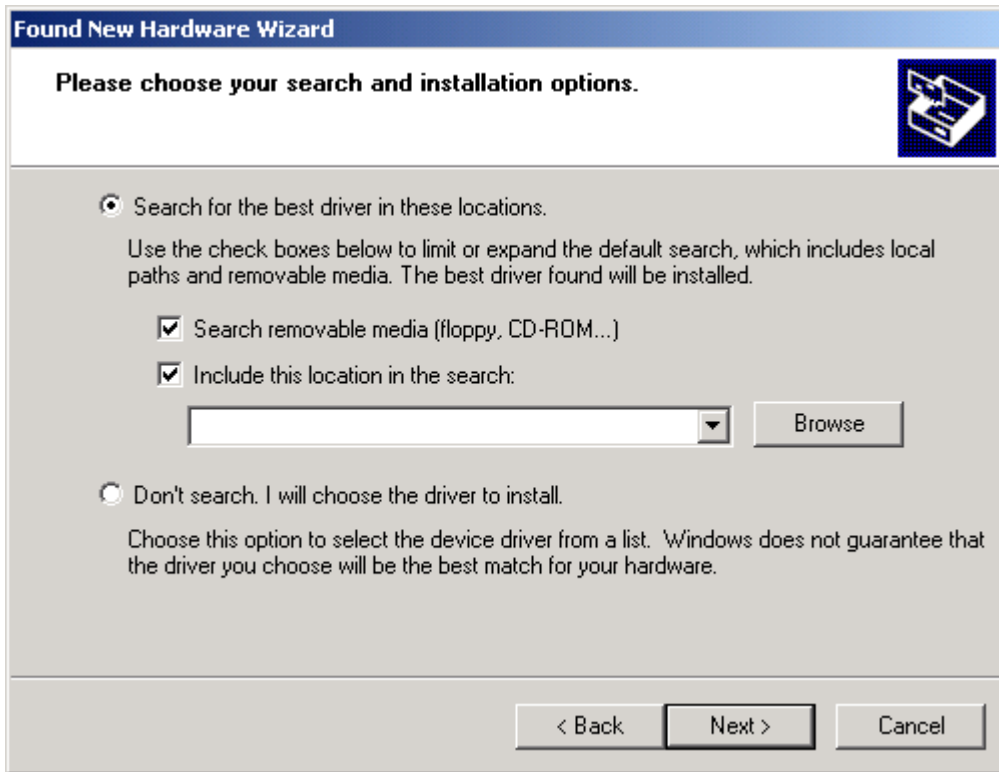
Windows Driver Installation

On Windows, follow the steps below to finalize the installation.

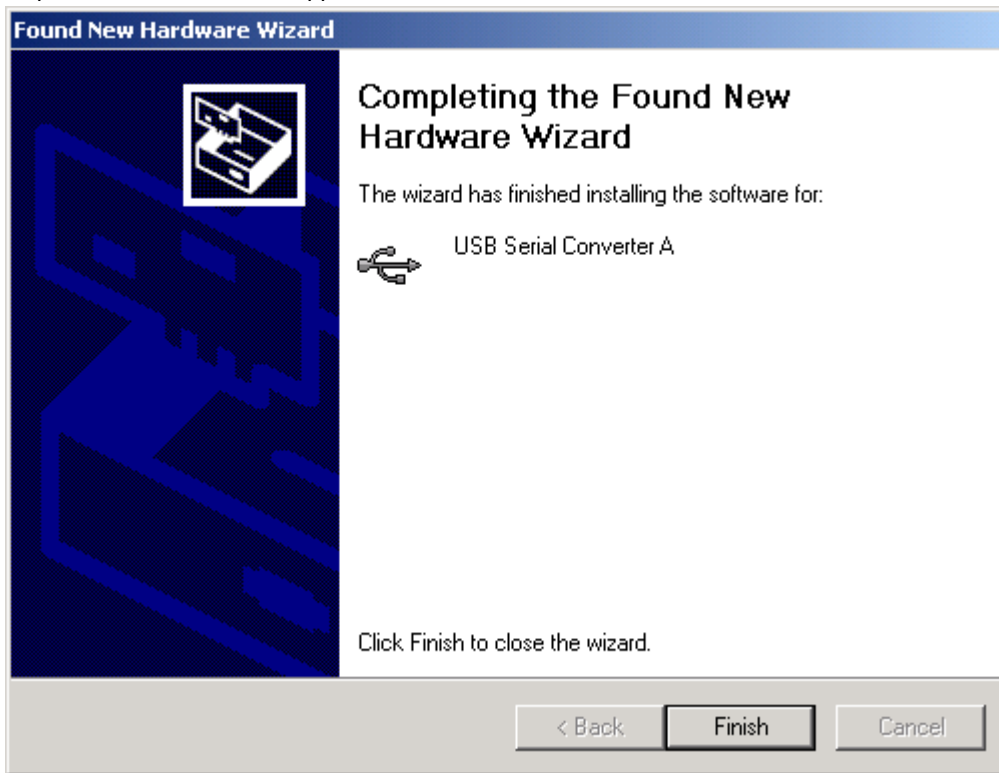
1. Connect the USB cable between the device (manager or mote) and your computer. If the Found New Hardware Wizard appears, go to step 2.
If the Found New Hardware Wizard does not appear, do the following:
 1. Ensure that the port is functional, and that the device is connected correctly. If the Wizard still does not appear, open the Windows Device Manager to see how Windows has recognized the device.
 2. If a device named "Dust Interface Board" is listed as an unknown device (yellow icon), right-click the device and select **Update Driver**. This displays the Found New Hardware Wizard.
 3. Go to step 2.
2. In the Wizard, click the option to **Install from a list or specific location** and click **Next**.




3. Select the box to **Include this location in the search**. Then, use the **Browse** button to navigate to your desktop, and click **Next**.



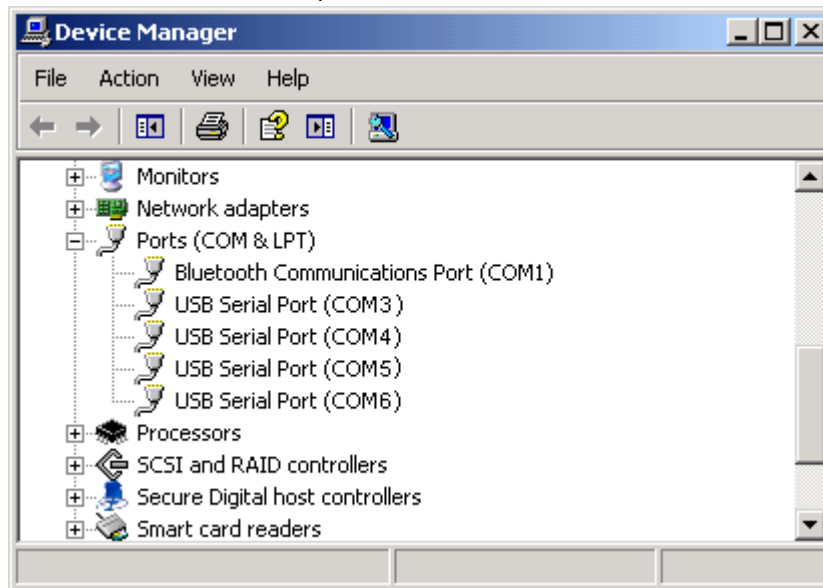
4. After the Wizard installs the software, click **Finish**.
5. When the Found New Hardware Wizard reappears, repeat steps 2 through 4 to continue the installation. Repeat these steps each time the Wizard appears.



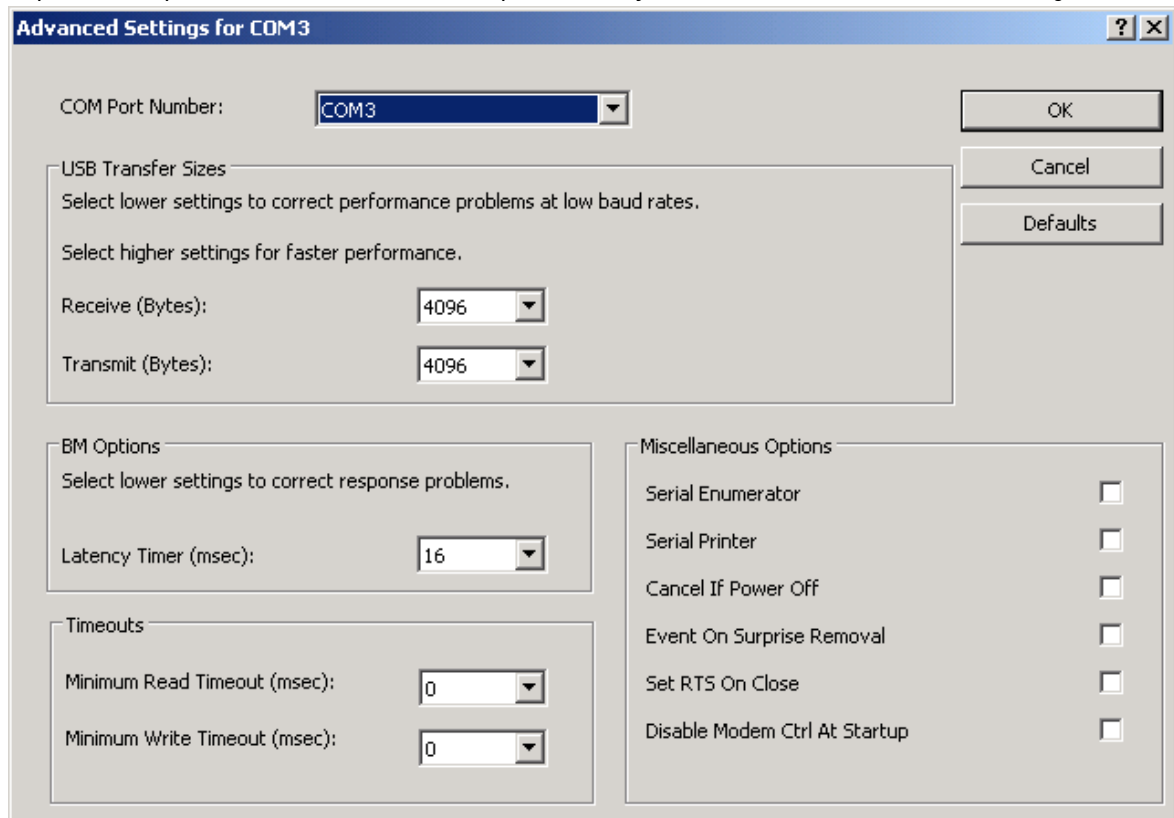
 Because of the way Windows works, you may be prompted to go through the Wizard up to eight times to complete the installation and mapping of the USB port. The manager will install a total of four virtual serial ports, along with the USB devices to control them.

6. When the installation and mapping of the USB ports is complete, open the Device Manager to find out the COM port numbers that have been assigned to the virtual serial ports.
 1. Choose the **Control Panel** from the Start menu.
 2. Open the **System** folder.
 3. Click the **Hardware** tab and click Device Manager.
 4. Open **Ports** to see the COM ports.

You should see four new COM ports in the Device Manager.
 5. Make a note of the four COM port numbers.



7. Configure the following **Advanced Settings** for each of the four new COM ports:
 1. Right-click a COM port and click **Properties**.
 2. Click the **Port Settings** tab, and then click **Advanced**.
 3. Deselect the **Serial Enumerator** option, and click **OK**.
 4. Click **OK** to return to the Device Manager.
 5. Repeat this step for each of the four new COM ports. When you are finished, close the Device Manager.



4.2.4 Interfacing to the Manager

Command Line Interface (CLI)

The VManager console application provides a command line interface that is intended for human interaction with the manager. System parameters such as Network ID can be configured through the CLI. System and network status information can also be retrieved via this interface. CLI access is protected with username/passwords that should be changed prior to deployment.

✔ For more detailed information on connection to the CLI and the commands that are available, refer to the [SmartMesh IP VManager CLI Guide](#).

Application Programming Interface (API)

The VManager API provides a programmatic interface for interacting with the network. External applications use the API to communicate with the manager. API access is protected with a username/password that should be changed prior to deployment.

Several tools are available for experimenting with and communicating with the manager API:

- The VManager API Explorer
- The VManager Python client

Throughout the documentation, VManager API calls are given in Swagger UI format, e.g. *POST /network/networkId*

✔ For more detailed information on connection to the API and the commands that are available, refer to the [SmartMesh IP VManager API Guide](#).

Network ports

The VManager host requires several open network ports:

- TCP port 8888 for HTTPS access to the VManager API
- TCP port 9101 for TLS access to the AP bridge software
- TCP port 22 for ssh access to the host console and VManager CLI
- UDP port 123 for NTP time synchronization

Authentication

The VManager console and API both require authentication against the VManager user database. The VManager console requests user credentials when it is started. Authentication for the VManager API is described in the API Guide.

The VManager provides two privilege levels: a user with *Viewer* privileges can view the network state and configuration. A user with *User* privileges can view the network state and edit the configuration, including adding new users. The default user credentials are:

- username: **dust**
- password: **dust**

The VManager maintains a list of active sessions. When the maximum number of sessions is reached, older (based on last activity) sessions will be removed. Sessions can be expired due to inactivity.

4.2.5 Configuration and Usage

Network ID

Networks are built using a unique Network ID that is configured both at the manager and at each mote prior to installation. The default Network ID for all SmartMesh IP software and devices is set to 1229 and can be changed. Overlapping installations can be arranged into separate networks by assigning unique Network IDs to the motes and manager in each network. Motes won't communicate with motes or a manager that are using a different Network ID.

The Network ID stored in the Manager configuration can be updated through the *PUT network/config* command, or the CLI `config set network` command. The manager must be restarted to use the new Network ID.

If a network has already been built with a given Network ID, the *POST network/networkId* command can be used to change the Network ID for both the manager and all motes currently joined. The *POST network/networkId* command reliably pushes a new Network ID to all the motes in a network. This operation may take several minutes to complete.

Network Topology

By far the most common topology used with SmartMesh IP is Mesh. This topology type offers the most resilience and data reliability by adding path diversity.

However, other topology types are available on the VManager, and can be set with the following command from within Console (or with a similar network configuration through the VManager API):

```
> config set network topologyType=TYPE
```

TYPE can be one of the following values:

- MESH (default): Motes are allowed to be parents to other motes and the manager will add bandwidth based on a mote's children's requirements.
- STAR: Only Access Point Mote (APM) devices are allowed to be parents. Motes that cannot hear an APM will never join the network.
- EVENT: Motes are allowed to be parents to other motes but do not get additional bandwidth based on their children's requirements. This topology type is suitable for deep networks of eight or more hops, see the Deep Network Application Note for more details.

Other Important Parameters

Other than Network ID, most other network configuration parameters can be left to default values unless a network must be built to service very specific needs.

Stored vs Active Configuration

The manager distinguishes stored configuration information (displayed and managed via the `config` CLI commands and `config` API resources) from the currently used configuration (displayed via the `show` CLI commands and `info` API resources). The manager makes this distinction so that configuration can be edited separately from applying it to an active system. For example, were you to look at the current system configuration via `show system`:

```
$> show system
System information:
  System start: 2015-12-04 15:13:42.433, Uptime:    3-23:15:12
  sysName:     Thermall
  location:    Unit5
  cliTimeout:  0 minutes
```

To change a parameter, one must first modify the stored configuration:

```
$> config set system cliTimeout=60
Done
```

Then you need to apply the stored configuration for the change(s) to take affect:

```
$> config reload system
Done
$> show system
System information:
  System start: 2015-12-04 15:13:42.433, Uptime:    3-23:15:12
  sysName:     Thermall
  location:    Unit5
  cliTimeout:  60 minutes
```

In the above example, only the system parameters are reloaded. If parameters of other modules are changed, the arguments to the `reload` command should be changed as well.

Refer to the [SmartMesh IP VManager CLI Guide](#) and [SmartMesh IP VManager API Guide](#) for details of the commands used.

4.2.6 Network Activity

Network Structure and Formation

The SmartMesh IP Manager performs automatic network management operations to maintain the health of the network as well as optimize it for the lowest power consumption and highest reliability. The manager also dynamically makes changes to the mesh as conditions in the network vary, such as path stability changes due to interference, addition and removal of service requirements, and addition and removal of devices. The customer applications never have to get involved in the management aspects of the network.

If motes are powered up in the vicinity of a manager, they will start joining almost immediately. The AP Mote advertises on average twice per second and motes, once joined, advertise once every two seconds. All motes will advertise except for those explicitly designated "non-routing".

In order for a mote to join a particular manager's network, the mote must be assigned the Network ID used by the manager. The manager and mote exchange several messages in the join process to establish security keys and proper network routing. The mote goes through several state changes during the join process, ending in the **Operational** state. The manager will detect if a mote leaves the network and will update the mote's state to **Lost**.

An external application can keep track of motes joining and leaving the network by listening to *Event* notifications.

Network Health

The motes and manager each keep track of network and device statistics. The motes generate *Health Reports* that are sent to the manager, who aggregates the resulting statistics on a per-mote and network wide basis.

Statistics queries are available through the following API commands:

- *GET /network/info* provides network-wide statistics.
- *GET /motes/m/{mac}/info* provides statistics accumulated for a mote.

From the CLI, the `show network` command shows the current (active) network configuration and network wide statistics.

```

Network configuration:
  networkId:          115
  topologyType:      MESH
  downFrameMultiplierDelay: 3600000
  ccaMode:           False
  ipAddrPrefix:      FE80::
  basePkPeriod:      15000
  downFrameMultiplier: 1
  joinSecurityType:  COMMON_SKEY
  minServicePkPeriod: 100
  downFrameSize:     512
  numParents:        2
  channelList:       32767
  upFrameSize:       1024
Network statistics:
  Network start time: 2015-12-15 11:39:45.035, uptime 1-01:40:21
  Live motes:        13
  Reliability:       100.000% (0 lost, 129257 total)
  Avg Latency:       1193 ms
  Path stability:    78.620%
  Advertising:       On
  Queue (net/user): 0/0
  Current frame size: 512

```

And the `show mote` command shows the current state, statistics, and neighbor information for the specified mote:

```

$> show mote 2

MOTE #2, MAC: 00-17-0D-00-00-12-34-56
  Version: 91.4.2.1 (stack 1.3.2.1)
  State: Oper, Hops: 1.0, State time: 1-01:55:18, Age: 1
  Power: 65534 (Routing)
  Power Cost: Max 65534, FullTx 65, FullRx 65, Used 817
  Capacity: 200 links, 31 neighbors
  Number of neighbors (parents, children) : 12 (1, 1)
  Bandwidth total / descendants (requested) : 929 / 969 (27840)
  Number of links total, TX / RX / requested: 56, 24 / 23 / 1

Statistics:
  Reliability: 100.000% (0 lost, 3379 total)
  Avg Latency: 600 ms, 3712 ms est. to mote
  Voltage: 3237 mV
  Charge consumed: 11935 mC

Neighbors:
  # 1 parent Q:66 links:24 rssi:-62/-59 Ready
  # 12 child Q:98 links:23 rssi:-24/-24 Ready

```

See the [SmartMesh IP VManager CLI Guide](#) for details on the meanings of the various fields.

The manager's statistics can be reset using the `POST /network/clearStats` API or the CLI `exec clearStats` command.

Health Reports

Motes periodically report statistics to the manager in health reports. The manager uses the health reports to determine a mote's neighbors and path stability for network optimization. The manager accumulates some lifetime statistics but does not store individual health reports.

Health reports are available as notifications via the API, so an external application connected to the manager can subscribe to health report notifications and track network statistics in detail over time.

Optimization


Based on information in the health reports, the manager continuously works on improving the network. For each mote, the manager considers how the existing parents compare to other neighbors discovered by the mote. If one existing parent looks better than the other, or if a discovered but unused neighbor looks like it would make a better parent, the manager will add a link from the mote to the better parent. This is called the *optimization add* cycle. Comparing the parents involves comparing a scoring function that attempts to minimize mote energy consumption while spreading traffic across the busiest motes. During the add cycle, the manager often has to guess at which paths are going to be best for the network and tries them out by adding single links to each mote.

After an hour, the manager has collected four health reports from the mote with the new parent and then has sufficient data to quantify which parents are best. Any mote with an extra link at this stage will delete the link to its worst parent, again by the same scoring metric. This is called the *optimization delete* cycle. The process continues for the life of the network, alternating add-delete-add-delete, each one hour apart. Running continuously allows the network to adapt to any changes in the environment.

4.2.7 Communicating with Motes

The manager API client application can communicate with motes and with mote API clients (e.g. a microcontroller) using the manager's `POST /motes/m/{mac}/dataPacket` or `POST /motes/m/{mac}/ipPacket` API to send packets to a mote and receives upstream data from motes by subscribing to data notifications using the `GET /notifications` command.

Each packet in the network contains source and destination addresses and source and destination ports. Ports are used in the TCP/UDP sense of providing efficient filtering for messages destined for a particular service. Refer to the SmartMesh IP Mote->Communication->UDP Port Assignment section in the [SmartMesh IP User's Guide](#) to decide which ports to use for the application.

 The difference between the *dataPacket* and *ipPacket* command is that in the latter the client supplies the 6LoWPAN header, allowing for more flexibility in source destination addressing, use of other transport besides UDP, etc. Not all possible combinations of the 6LoWPAN header fields are supported by the mote. See the [SmartMesh IP VManager API Guide](#) for details on manager commands.

4.2.8 Access Control

ACL Management

Motes use a join key to encrypt the initial join request when joining a network. If the join key used by the mote does not match the mote's join key configured on the manager, the manager will not be able to decrypt the join message and will not allow the mote to join the network. By default, motes and the manager ship with a common join key set to a default value - see the [The SmartMesh IP Network](#) section of this guide for more details on security.

In its simplest, and least secure mode of operation, the manager will accept any mote which uses a common join key. For added security, the manager maintains an Access Control List (ACL) which associates a mote's MAC address with a mote-specific join key. This list is normally created prior to network formation, but can be updated and modified at run time as new devices are added to the network. In the most flexible configuration, motes can join with a mix of common join key and unique join keys. If the *common_skey* join mode is set in the network configuration, then a mote can join with either the common join key or an entry in the ACL. Additionally, a mote can be prevented from joining by putting its MAC address on a blacklist called the Deny Control List (DCL). If a mote is entered in both the DCL and the ACL, then the ACL takes precedence. If the *unique_skey* join mode is set in the network configuration, then each mote must be on the ACL to join and can no longer join using the common join key.

In the event that a join key for a mote needs to be changed, for example if the initial join at deployment is done with common join key, the `POST /motes/m/{mac}/joinkey` API command or the `exec exchJoinKey` CLI command can be used. This is an operation in which the manager pushes a new join key to a mote and updates the ACL. If the mote doesn't acknowledge the update, then the manager will not update its ACL. Alternatively, a mote can be added manually to the ACL through the `PUT /acl/config/{mac}` command in the API or the `config set acl` command in the CLI.

The most straightforward procedure for setting up ACL for all the motes in a network is with the following steps:

- Let motes join the network using the common join key
- For each mote in the network, use the `POST /motes/m/{mac}/joinkey` command to set an ACL entry for each mote with MAC address {mac}



The maximum number of entries in the ACL is 50,000 as of version 1.1.0

The maximum number of entries in the DCL is 5,000

4.2.9 Redundancy

AP Bridge:

A system with only one AP Bridge may be perfectly suitable under normal operation, but if it fails, resets, or loses connection to the VManager, the entire network will be restarted. Redundancy is easily achieved by installing at least one more AP Bridge in the same area. If either fails, all network traffic will be automatically routed through the remaining AP Bridge. A system consisting of multiple AP Bridges properly meshed (each within reach of at least one other AP Bridge) is completely redundant.

Special consideration should be given to a network built in GPS mode, namely a network consisting of one or more GPS AP Bridges. Such a network can be built with any number and mix of GPS and non-GPS APs, however there must always be at least one GPS AP Bridge active at any time. For example, if a network is built with a single GPS and 5 non-GPS AP Bridges, the network will be completely reset and rebuilt if that one GPS-enabled unit fails. For redundancy, it is best practice to install 2 co-located GPS-enabled AP Bridges.

Manager:

VManager redundancy is accomplished using Linux-HA to provide for failure detection and hot swap capability.

This feature is not supported in the 1.1.0 version.

4.2.10 Over the Air Programming (OTAP)

An external application can communicate with motes through the manager and upgrade firmware on the devices – this is called Over-The-Air-Programming (OTAP). The SmartMesh SDK provides a reference application called OTAP Communicator.

The basic steps of an OTAP operation are as follows:


1. Updated firmware is provided in the form of a `.otap2` file.
2. A handshake with all motes determines which motes can accept this update (the receive list), and prepares them for update.
3. The OTAP file is divided into packets and sent to all devices - only devices on the receive list will do anything with the file.
4. Process queries all motes on the receive list to verify that the image was received and is valid. Steps 2 and 3 are repeated until all (eligible) motes have received the file.
5. Sends the commit message to all motes that received the file. This will cause them to reprogram their flash with the new firmware.
6. Motes are reset and rejoin the network using the new firmware.

4.2.11 Restoring Manager Factory Default Settings

The manager can be restored to factory settings in one of two ways:

- Using the `config restore` CLI command
- Using the `POST /config/restore` API command

Restoring also clears the access control and deny control lists. The table below lists defaults for the configuration parameters affected by restoring to factory settings.

 The manager must be restarted (from a Linux prompt) using the `smctl restart` command after restoring defaults.


Parameter	Comment	Default
networkId	Network ID	1229
upFrameSize	Upstream Frame size (slots)	512
downFrameSize	Downstream Frame size (slots)	512
downFrameMultiplier	Downstream frame multiplier for steady state. The downstream frame is extended from <i>downFrameSize</i> to <i>downFrameSize</i> * <i>downFrameMultiplier</i> after the network has formed	1
downFrameMultiplierDelay	Time from first mote join until switching to use the downstream frame multiplier (ms)	3600000
numParents	Minimum number of parents	2
ccaMode	Mode used for clear channel assessment	0 = False (Off)
channelList	Bitmap of used channels (b0 = IEEE channel 11)	7FFF = 32767
basePkPeriod	Base (minimum) Bandwidth to allocate to each device (ms)	15000 (in ms/packet)
commonJoinKey	Common Join Key	44 55 53 54 4E 45 54 57 4F 52 4B 53 52 4F 43 4B (hex)
ipAddrPrefix	IPv6 Address Prefix	FE80::
minServicePkPeriod	Minimum data interval (fastest service) allowed (ms)	100

joinSecurityType	Join Security model	COMMON_SKEY (Common join key)
topologyType	Topology Type	MESH
userId	UserId of the default user	dust
password	Password of the default user	dust
privilege	Privilege level of the default user	USER
sysName	System name string	(no default)
location	System location string	(no default)
cliTimeout	Timeout for logging out an inactive console login	0 (no timeout)

4.2.12 Channel Blacklisting

The default behavior for SmartMesh Networks is to blacklist only the sixteenth channel (2480 MHz) to comply with requirements in the United States (as regulated by FCC) and Canada (as regulated by IC). The `PUT /network/config` command is used to specify a bitmap of which channels are used, and the configuration must be reloaded by restarting the manager. Channel blacklisting is a network-wide property.

Channel 0 (2405 MHz) corresponds to IEEE channel 11, and Channel 15 (2480 MHz) corresponds to IEEE channel 26.

- 
 The user is responsible for ensuring that the number of allowed frequencies conforms with local RF regulations. Although the network may operate on as few as five channels, it is recommended that the network run on as many channels as possible for greater resiliency and more overall bandwidth.

4.2.13 Database Backups and Restores

The VManager provides command line tools for generating and restoring backups of the configuration database.

Backup


To perform a backup, run the `db_save.py` tool from the Linux command line. In the example, the tool creates a backup file named `backup.db_backup`.

```
$ cd /opt/dust-voyager
$ python bin/db_save.py --api-proto tcp --type backup --file backup.db_backup
```

Restore

To perform a restore, the backup file must be manually moved into place and the existing configuration database removed. The restore should be performed while the VManager is shut down. After the configuration database files are replaced, when the VManager starts, it will restore the configuration data from the `configdb.db_backup` file.

```
$ cd /opt/dust-voyager
$ smctl stop
.. wait for processes to stop ..
$ sudo -u dustadmin cp backup.db_backup var/db/configdb.db_backup
$ sudo -u dustadmin rm var/db/configdb.db_data
$ smctl start
```

 The `var/db` paths in the commands above are relative to the `/opt/dust-voyager` directory.

5 SmartMesh Glossary

Access Point Bridge - A device containing an Access Point Mote and Access Point Controller with a TCP/IP connection to the manager.

Access Point Controller - A software process that handles interaction between the Access Point Mote and the VManager via a TCP/IP network.

Access Point Mote (or APM) - The APM connects the wireless network to the manager. Converts wireless MAC packets to wired Net packets and vice versa.

Acknowledgement (or ACK) - A frame sent in response to receiving a packet, confirming that the packet was properly received. The ACK contains the time difference between the packet receiver and sender.

Advertisement - A frame sent to allow other devices to synchronize to the network and containing link information required for a new mote to join.

API Client - A software program or device that connects to the manager or mote's application programming interface (API).

ASN - Absolute Slot Number. The number of timeslots that have elapsed since the start of the network (WirelessHART) or 20:00:00 UTC July 2,2002 (IP if UTC is set before starting the network) .

Authentication - The cryptographic process of ensuring that the packet received has not been modified, and that it originated from the claimed net layer sender.

Backbone - A feature in a SmartMesh network that allows motes to share a fast superframe to enable low-latency alarm traffic. (Only available in the embedded SmartMesh IP manager and SmartMesh WirelessHART manager)

Bandwidth - The capacity of a mote to transmit data, usually expressed in packets/s.

Base Bandwidth - The bandwidth each mote in a network gets without having to request a service.

CCM* - Counter mode with CBC-MAC. The authentication/encryption scheme used in Dust products. See the application note "SmartMesh Security" for a more detailed description.

Channel - The index into a list of center frequencies used by the PHY. In 802.15.4, there are 16 channels in the 2.4-2.4835 GHz Instrumentation, Scientific, and Medical (ISM) band.

Channel Hopping - Changing channel between slots, also called "slow hopping" to distinguish from PHY's that change channel within a message, such as Bluetooth.

Channel Offset - A link-specific number used in the channel calculation function to pick which channel to use in this ASN.

Child - A device that receives time information from another mote is its child. A child forwards data through its parent.

CSMA - Carrier Sense Multiple Access. A communications architecture where an unsynchronized transmitter first senses if others are transmitting before attempting its transmission.

Commissioning - The act of configuring motes for use in a deployment, typically by setting Network ID, join key, and other joining parameters.

Discovery - The process by which motes find potential neighbors, and report that information back to the manager.

Downstream - The direction away from the manager or wired-side application and into the mesh network.

Embedded Manager - A single-chip LTC5800-based manager that combines manager and AP mote functions, suitable for networks of 100 motes or less. See VManager for larger networks.

Encryption - The cryptographic process of converting payload information into a form indistinguishable from random noise, such that it can only be read by the intended recipient.

Frame (or packet) - Information sent by the PHY layer. Typically containing per-hop (MAC) and end-to-end (Net) layer headers and a payload.

Gateway - The device in a WirelessHART network responsible for abstracting the wired HART data model from the WirelessHART mesh implementation.

Graph - A numbered routing element included in the WirelessHART net and IP mesh layer headers which tells a mote where to send each packet. Superframes and slotframes also have a graph ID.

Graph route - A route where only the graph ID is specified in the packet header. A packet can follow the graph route through multiple motes to its destination. Compare with source route.

Guard time - The time a device listens in advance or after the expected packet arrival time to allow for imperfect synchronization between devices.

Health report - A packet sent by a mote conveying its internal state and the quality of its neighbor paths. Health reports are used by the manager in optimization and diagnostics.

Idle listen - A slot where the receiving device wakes up to receive a packet but the transmitting device does not send one. About two-thirds of all receive slots end up as idle listens in a typical network.

Joining - The sequence of handshakes between a new mote and a manager to bring the mote into the network. It begins with a mote presenting an encrypted request and ends with link and run-time security credential assignment.

Keep alive - An empty packet sent to keep devices synchronized after a timeout during which no data packets have been sent on a path. This timeout is typically 30 seconds in WirelessHART and 15 seconds in IP.

Latency - The time difference between packet generation and arrival at its final destination.

Leaf - A mote that presently has no upstream RX links, and thus no children. Also called a leaf node or leaf mote.

Low Power Border Router (LBR) - A logical or physical device which converts 6LowPAN (IPv6 for Low power Wireless Personal Area Networks, RFC 4944) packets into IPv6 packets. Also called an Edge Router.

MAC - The Medium Access Control layer. Technically a sublayer of the Data Link Layer (OSI layer 2), but typically used interchangeably in Dust documentation.

Manager - The device or process responsible for establishing and maintaining the network.

Master (mode) - a mode of mote operation where the mote automatically joins the network for which it was configured, the serial API is disabled, and a resident application allows for interacting with some onboard I/O.

Mesh - A network topology where each mote may be connected to one or more motes.

Mote - A device which provides wireless communications for a field device to transmit sensor or other data. The basic building block of a network.

Multi-hop - A network where one or more motes has no path to the access point. Data packets may sometimes take multiple hops from source to destination.

Multipath - A radio signal that is the superposition of many reflected signals. Used as an adjective to describe phenomenon where signal level can vary dramatically with small environmental changes, *i.e.* multipath propagation, or multipath fading.

NACK - A special ACK frame sent in response to receiving a packet, stating that it was correctly received but NOT accepted for forwarding.

Neighbor - A mote in range of the mote in question.

Non-routing - A mote that has been configured to not advertise. A non-routing mote never forwards packets along a graph or has children.

On-Chip SDK - A Software Development Kit (SDK) for writing applications (API clients) that run on the mote Cortex-M3 chip.

Optimization - The manager process of taking health report information and using it to modify the network to minimize energy consumption and latency.

Packet - Also called a frame. The variable sized unit of data exchange.

Parent (or time parent) - A device that serves as a source of time synchronization. In Dust networks, a mote's parent is one hop closer to the AP. A parent forwards a child's data towards the manager.

Path - The potential connection between two motes. A path that has assigned links is a used path. One that has been discovered but has no links is an unused path.

Path stability - The ratio of acknowledged packets to sent packets between two motes. Each of the two motes keeps a separate count of the path stability denoted by A->B and B->A. A path where the two motes have significantly different counts of path stability is called an asymmetric path.

Pipe - A feature in a SmartMesh WirelessHART network that enables rapid publishing to and/or from a single target mote.

Provisioning - The number of links assigned by the manager per packet generated by the mote to allow for imperfect stability. Default provisioning is 3x meaning that on average each packet has three chances to be successfully transmitted before the mote starts to accumulate packets.

Publishing rate (Burst rate) - The rate at which an mote application transmits upstream data. In WirelessHART the term burst rate is equivalent to Publishing rate.

PHY - The physical layer, *i.e.* the radio.

Reliability - The percentage of unique packets received relative to the number generated.

Route - The motes that a packet passes through between source and destination, *e.g.* a packet from mote 3 might use the route 3-2-6-AP. Because of the graph routing used upstream, packets originating at the same mote randomly take a variety of routes.

Schedule - The collection of superframes and links in the network, particularly those organized for a particular purpose.

Services (or Timetables) - The process of requesting and receiving (or not) task-specific bandwidth.

Slave (mode) - a mode of mote operation where the mote requires an application to drive its API in order to join the network.

Slotframe or Superframe - A collection of timeslots with a particular repetition period (length) and labeled with a Graph ID.

Source route - A route where every hop between source and destination is explicitly specified in the packet header. Dust networks only use source routing for downstream packets.

Star - A network topology where all motes only have a connection to the the access point (in ZigBee, the PAN coordinator) and are non-routing leaves.

Star-mesh - A network topology where motes form stars around routers, where the routers may have one or more neighbors through whom they can forward mote data.

Statistics - The aggregated information about network topology and performance constructed from the raw mote health reports.

TDMA - Time Division Multiple Access. A communications architecture where packetized information exchange only occurs within timeslots and channel offsets that are assigned exclusively to a pair of devices.


Timeslot - A defined period of time just long enough for a pair of motes to exchange a maximum-length packet and an acknowledgement. Time in the network is broken up into synchronized timeslots.

Upstream - The direction towards the manager or wired-side application from the mesh network.

VManager - A manager process designed to run on a Linux server, suitable for networks of any size. See also Embedded Manager.

ZigBee - A single channel, CSMA network protocol.

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 Linear Technology 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 Linear Technology.

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.

Linear Technology 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.

Linear Technology 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. Linear Technology customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify and hold Linear Technology 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 Linear Technology was negligent regarding the design or manufacture of its products.

Linear Technology 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.

Linear Technology does not warrant or represent that any license, either express or implied, is granted under any Linear Technology patent right, copyright, mask work right, or other Linear Technology intellectual property right relating to any combination, machine, or process in which Linear Technology products or services are used. Information published by Linear Technology regarding third-party products or services does not constitute a license from Linear Technology 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 Linear Technology under the patents or other intellectual property of Linear Technology.

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

© Linear Technology Corp. 2012-2016 All Rights Reserved.