SmartMesh WirelessHART Application Notes

Table of Contents

1 Revision History

Revision	Date	Description
1	03/18/2013	Initial Release
2	07/18/2013	Added "What is Packet ID and why do I Need it?" Application Note
3	10/22/2013	Minor corrections
4	03/12/2014	Added "Monitoring SmartMesh WirelessHART Network Health" and "Using the Powered Backbone to Improve Latency" Application Notes
5	04/04/2014	Updated "Using the Powered Backbone to Improve Latency" Application Note Clarified how Q is calculated
6	04/11/2014	Added "Building Deep WirelessHART Networks" Application Note

2 Application Note Summary

This document contains a collection of Application Notes about the SmartMesh WirelessHART product family:

- How to Evaluate WirelessHART Network and Device Performance - Measuring several performance metrics.
- Data Publishing for SmartMesh WirelessHART - Low-level descriptions of how to use the mote's API for sending data.
- Using WireShark to Troubleshoot HART Manager Connections - An external application helpful for debugging the XML interface.
- Testing for WirelessHART Certification - Basics of getting a device certified for the WirelessHART standard.
- Custom Configuring a Manager for More Downstream Bandwidth - Trading off power for downstream bandwidth in smaller networks.
- Manager Linux Shell Commands - Useful commands for configuring the SmartMesh WirelessHART manager from the Linux shell.
- How to Interact Programmatically with the Manager Serial 1 port - Python example script for driving the Serial 1 port programmatically.
- Building a WirelessHART Compliant Device - Discusses sensor application requirements for designs that need to be WirelessHART compliant.
- Using the Powered Backbone to Improve Latency - Description of several backbone use cases.
- Monitoring SmartMesh WirelessHART Network Health - Automated checks to ensure your network is performing well.
- Building Deep WirelessHART Networks - Settings to use for networks up to 32 hops deep.

The following Application Notes apply to both SmartMesh IP and WirelessHART families. Differences between the products, if any, are highlighted:

- Planning A Deployment - High-level considerations prior to deploying a network.
- Predicting Network Health - Initial post-deployment assessment tips.
- Common Problems and Solutions - General troubleshooting advice.
- Changing Provisioning Factor to Increase Manager Throughput - Decreasing per-mote bandwidth to support more total traffic in high-stability conditions.
- Debugging Congested Networks - Understanding impact on network operation when mote queues are full and tips on remedying such situations.
- Identifying and Mitigating the Effects of Interference - Graphically representing network statistics to notice interference-related problems.
- Obtaining Accurate Timestamps - How to use the network time synchronization properly for timestamping packets.
- Using Multiple Managers to Build Large Networks - Considerations for deployments exceeding the mote limits of a single manager.
- Using the SmartMesh Power and Performance Estimator - Examples on using the spreadsheet for predicting power and latency in a variety of networks.
- What to Expect with Motes That Move - Details on the behavior of a mote moving to a different location in the same network.
- Migrating Motes Between Networks - Details on how to migrate motes between different networks.

- Configuring a Network for Bounded Data Latency - Adding bandwidth to a network to decrease the packet latency.
- Network Coexistence - Features that allow multiple networks to operate in the same radio space.
- How to Choose a Join Duty Cycle - Considerations for trading off power and join speed.
- SmartMesh Security - A description of the security used in all SmartMesh networks.
- Using the TestRadio Commands - A description of how to use APIs for testing radio performance or certification.
- Best Practices to Limit Average Current During Peak Periods - Guidelines to follow in order to keep average current down when booting or writing flash.
- Methodology For Pilot Network Evaluation - Overview of deployment and statistics collection for network pilots.
- What is Packet ID and why do I Need it? - Details on the single bit used to provide reliable call and response with the Sensor Processor.

3 Application Note: How to Evaluate WirelessHART Network and Device Performance

3.1 Performance Evaluation

OK, you just got a SmartMesh WirelessHART starter kit - now what? This app note describes a number of tests you can run to evaluate specific performance characteristics of a SmartMesh network. It assumes that you have installed the SmartMesh SDK python tools, and the necessary FTDI drivers to communicate with motes. It also presumes you have access to the Manager and Mote CLI and API documentation.

First thing is to connect to the manager (PM2511 or LTP5903CEN-WHR) as described in the "WirelessHART Manager CLI Introduction" section of the SmartMesh WirelessHART Manager CLI Guide and log into the manager's CLI as described in "Logging on to the Manager CLI." You will also need to connect a DC9006 Eterna Interface Card to one or more DC9003A-C motes in order to access the mote's CLI to configure it (see the SmartMesh WirelessHART Mote CLI Guide for details).

3.2 Join Behavior

How fast does a mote join? Mote join speed is a function of 6 things:

- Advertising rate - over which the user has little control other than mote density
- Join duty cycle - how much time an new mote spends listening for a network versus sleeping
- Mote join state machine timeouts - there is no user control over these timeouts in SmartMesh WirelessHART
- Downstream bandwidth - affects how quickly motes can move from synchronized to operational
- Number of motes - contention among many motes simultaneously trying to join for limited resources slows down joining
- Path stability - over which the user has little or no control

The join process is broken up into three phases: search, synchronization, and message exchange.

- Search time is determined by advertising rate, path stability, and mote join duty cycle. This can be 10's of seconds to 10's of minutes, largely depending upon join duty cycle
- Synchronization is determined by mote state machine timeouts - this period is 30 seconds in SmartMesh WirelessHART by default. It can be adjusted by setting the *advtout* (default = 30 s) and *numjnpar (*default *= 3)* parameters using the `mseti` mote CLI command, but only in a network that does not need to comply with HART standards.
- Message exchange is determined by downstream bandwidth and number of motes (which compete for downstream bandwidth). This is a minimum of ~15 s per mote, and can be much slower.

So, for example, it may seem that to join a network quickly, you would set the join duty cycle to maximum. However, this may result in a lot of motes competing for downstream bandwidth, and a network may form more quickly with a lower setting. In the experiments below, you will examine the "knobs" you have to control search and message exchange phases.

3.2.1 Tradeoff Between Search Time and Average Current

An unsynchronized mote listens for manager and other mote advertisements to synchronize to the network. The fraction of time spent listening versus sleeping is called the *join duty cycle*. The join duty cycle is one-byte field that can be set from 0 (0.2%) to 255 (almost 100%) - the default is 5% or a setting of 13, which is the default mode. See the SmartMesh WirelessHART Mote CLI Guide for instructions on changing operating mode. Lower settings result in longer search times at lower average current, while higher settings shorten search but increase average current. Providing that at least one mote is advertising in the mote's vicinity, the **energy** dedicated to joining remains the same - join duty cycle only affects average **current**. If a mote is isolated, or the manager is not on, a mote could potentially exhaust its battery looking for a network, so join duty cycle gives the user control over the tradeoff between speed when a network is present and how much energy is used when it isn't.

The time for a mote to synchronize and send in its join request is the first visible sign of a mote joining a network. This can be seen on the manager by using the `trace` CLI command:

```
> trace motest on
```

When the manager sees the mote's join request, it will mark the mote as being in the **Negotiation1** (Negot1) state.

To see the effect of join duty cycle:

- Connect to the manager CLI and turn on the mote state trace as described above.
- Connect to the mote CLI by using an Eterna Interface Card (DC9006)
- Reset the mote - it will print out a reset message

```
> reset
HART Mote 1.0.0-104
```

this will cause the mote to begin searching for the network with it's default duty cycle of 5% (13). You should see a a series of notifications on the manager as it changes state.

```
Mote #2 changed state to Negot1
Mote #2 changed state to Negot2
Mote #2 changed state to Conn
Mote #2 changed state to Oper
```

It should take ~60 seconds on average for a mote to synchronize and sent its join request. You may want to repeat several times (resetting and issuing a *join*) to see the distribution of synchronization times.

- Reset the mote and use the mote CLI to set the join duty cycle to 5% (0.05 * 255 = 13) - this is controlled through the mote CLI command `mseti joindc` command Join duty cycle persists through reset, so you only need to do this step once for each join duty cycle change. Measure how long it takes the mote to transition to the **Negotiating1** state, repeating the measurement as described above. It should be ~3 min on average.

```
> mseti joindc 13
```

- Reset the mote and use the mote CLI to set the join duty cycle to 100% (255). Measure how long it takes the mote to transition to the **Negotiating1** state. It should be <60 s on average.

```
> mseti joindc 255
```

Note: When the manager sees a join request from a mote that was previously operational, it will mark it "lost" first before promoting it to connected. It will appear as:

```
Mote #2 changed state to Lost
Mote #2 changed state to Negot1
```

3.2.2 Measuring Join Time

It takes a number of packets sent by the manager and acknowledged by the mote to transition a mote to the **Operational** (Oper) state where it can begin sending data. We can use the manager `trace io` CLI command to see these state transitions and measure how long they take. Reset the mote and watch it transition to **Operational**. Each line in the following trace represents one handshake between the mote and the manager:

```
> trace io on
```

```
> IO INP [07/03 12:30:03.867] AP-TIMEINFO asn=1380227 offset=4997
IO INP [07/03 12:30:38.538] SKJOIN src_mote=00-17-0D-00-00-30-00-70 dst_mote=63872 ttl=126 pr=0
tr=UnrlblResp.U#0 sec=SKHello c
Mote #2 changed state to Lost
Mote #2 changed state to Negot1
IO OUT [07/03 12:30:38.557] REG src_mote=63872 dst_mote=1 ap_link=255.65535 ttl=127 pr=11 tr=Rlbl#7
sec=SKData ts=12:30:38.556
IO OUT [07/03 12:30:38.562] SKACT src_mote=63872 dst_mote=2 ap_link=1.157 ttl=127 pr=11 tr=Rlbl#4
sec=SKHelloAck ts=12:30:38.56
IO INP [07/03 12:30:38.628] MNGR src_mote=1 dst_mote=63872 ttl=127 pr=0 tr=RlblResp.U#7 sec=SKData
count=15 | DevSt=0:0 | D_LIN
IO INP [07/03 12:30:39.966] AP Link Feedback 1:157:9
IO INP [07/03 12:30:42.628] MNGR src_mote=2 dst_mote=63872 ttl=126 pr=0 tr=RlblResp.U#4 sec=SKData
count=0 | DevSt=0:0 | W_SESS
Mote #2 changed state to Negot2
IO OUT [07/03 12:30:42.639] REG src_mote=63872 dst_mote=1 ap_link=255.65535 ttl=127 pr=14 tr=Rlbl#8
sec=SKData ts=12:30:42.638
IO OUT [07/03 12:30:42.644] ACT src_mote=63872 dst_mote=2 ap_link=1.157 ttl=127 pr=14 tr=Rlbl#5
sec=SKData ts=12:30:42.643  asn
IO INP [07/03 12:30:42.668] MNGR src_mote=1 dst_mote=63872 ttl=127 pr=0 tr=RlblResp.U#8 sec=SKData
count=16 | DevSt=0:0 | W_LIN
IO INP [07/03 12:30:45.086] AP Link Feedback 1:157:9
IO INP [07/03 12:30:50.518] MNGR src_mote=2 dst_mote=63872 ttl=126 pr=0 tr=RlblResp.U#5 sec=SKData
count=1 | DevSt=0:0 | W_FRAM
Mote #2 changed state to Conn
IO OUT [07/03 12:30:50.538] REG src_mote=63872 dst_mote=1 ap_link=255.65535 ttl=127 pr=12 tr=Rlbl#9
sec=SKData ts=12:30:50.537
IO OUT [07/03 12:30:50.544] REG src_mote=63872 dst_mote=2 ap_link=1.157 ttl=127 pr=12 tr=Rlbl#6
sec=SKData ts=12:30:50.543  asn
IO INP [07/03 12:30:50.568] MNGR src_mote=1 dst_mote=63872 ttl=127 pr=0 tr=RlblResp.U#9 sec=SKData
count=17 | DevSt=0:0 | W_LIN
IO INP [07/03 12:30:52.766] AP Link Feedback 1:157:9
IO INP [07/03 12:31:00.758] MNGR src_mote=2 dst_mote=63872 ttl=126 pr=0 tr=RlblResp.U#6 sec=SKData
count=2 | DevSt=0:0 | W_SESS
IO OUT [07/03 12:31:00.761] REG src_mote=63872 dst_mote=2 ap_link=1.157 ttl=127 pr=12 tr=Rlbl#7
sec=SKData ts=12:31:00.760  asn
IO INP [07/03 12:31:03.006] AP Link Feedback 1:157:9
IO INP [07/03 12:31:03.390] MNGR src_mote=2 dst_mote=63872 ttl=126 pr=0 tr=RlblResp.U#7 sec=SKData
count=3 | DevSt=0:0 | W_LINK
IO OUT [07/03 12:31:03.393] REG src_mote=63872 dst_mote=2 ap_link=1.185 ttl=127 pr=12 tr=Rlbl#8
sec=SKData ts=12:31:03.392  asn
IO INP [07/03 12:31:03.877] AP-TIMEINFO asn=1386228 offset=5363
IO INP [07/03 12:31:08.406] AP Link Feedback 1:185:5
IO INP [07/03 12:31:10.998] MNGR src_mote=2 dst_mote=63872 ttl=126 pr=0 tr=RlblResp.U#8 sec=SKData
count=4 | DevSt=0:0 | W_TIME
IO OUT [07/03 12:31:11.003] REG src_mote=63872 dst_mote=2 ap_link=1.185 ttl=127 pr=12 tr=Rlbl#9
sec=SKData ts=12:31:11.003  asn
IO INP [07/03 12:31:16.086] AP Link Feedback 1:185:5
IO INP [07/03 12:31:18.778] MNGR src_mote=2 dst_mote=63872 ttl=126 pr=0 tr=RlblResp.U#9 sec=SKData
count=5 | DevSt=0:0 | W_SESS
Mote #2 changed state to Oper
IO OUT [07/03 12:31:18.785] REG src_mote=63872 dst_mote=2 ap_link=1.185 ttl=127 pr=12 tr=Rlbl#10
sec=SKData ts=12:31:18.785  as
```

```
IO INP [07/03 12:31:21.206] AP Link Feedback 1:185:5
IO INP [07/03 12:31:23.868] MNGR src_mote=2 dst_mote=63872 ttl=126 pr=0 tr=RlblResp.U#10 sec=SKData
count=6 | DevSt=0:0 | W_ROU
```

Here it took ~45 s to transition. You should turn off the io trace at this point.

3.2.3 Effect of Downstream Bandwidth on Join

The rate at which the manager can send packets is a function of the downstream superframe size - the manager assigns the same number of dowstream links, regardless of superframe size, so a 4x longer superframe has 1/4 the bandwidth. By default, the SmartMesh WirelessHART manager uses a 256-slot superframe, and the set of frame sizes is called a *profile*, here P1. P2 changes the downstream frame to 2048 slots to save power in cases where little or no downstream traffic is expected. A longer frame means fewer downstream listens per unit time, so it will lower the average current for all motes, but also slow down the join process (and any other downstream data) for motes added later. While this has no effect on the time it takes to synchronize to the network (time to the **Negotiating1** state), it spaces out the other state transitions.

To see the effect of longer downstream superframe:

- Via manager CLI, confirm that the normal superframes are being used. Look for the line that shows the profile (*bandwidthProfile*):

```
> get network
netName:                myNet
networkId:              293
optimizationEnable:     true
maxMotes:               251
numMotes:               0
accessPointPA:          true
ccaEnabled:             false
requestedBasePkPeriod:  100000
minServicesPkPeriod:    400
minPipePkPeriod:        480
bandwidthProfile:       P1
manualUSFrameSize:      1024
manualDSFrameSize:      256
manualAdvFrameSize:     128
netQueueSize:           0
userQueueSize:          0
locationMode:           off
```

- Now you will adjust the downstream superframe multiplier to give a 2048-slot superframe - by setting the profile to P2:

```
> set network bandwidthProfile=P2
```

You will need to reset the manager for the profile to take effect.

- Confirm that the manager is using the longer downstream superframe:

```
> get network
netName:                myNet
networkId:              293
optimizationEnable:     true
maxMotes:               251
numMotes:               1
accessPointPA:          true
ccaEnabled:             false
requestedBasePkPeriod:  100000
minServicesPkPeriod:    400
minPipePkPeriod:        480
bandwidthProfile:       P2
manualUSFrameSize:      1024
manualDSFrameSize:      256
manualAdvFrameSize:     128
netQueueSize:           0
userQueueSize:          0
locationMode:           off
```

Manager CLI commands often use the term "frame" instead of the more formal "superframe." "Frame" is never used to indicate an 802.15.4 packet.

- Now power cycle the single mote again and note the time it takes for the mote to transition from **Negotiating1** to **Operational**. This should be ~8x the "fast" join time you saw above.

Path stability also affects join time, as worse path stability means more retries. Having a mesh architecture means that the effect of any individual path is minimized.

3.2.4 Network Formation vs Single Mote Join

The number of motes joining simultaneously also affects network formation time, as these motes must compete for limited join links and downstream bandwidth. To see this effect (this assumes you are using the longer downstream frame from above):

1. Configure the rest of your motes for 100% join duty cycle (as described above) - we aren't concerned with average current here
2. Reset a single mote, and let it join. Repeat this 10 times to get an average join time.

3. Reset all motes and note the time it takes for the network to form completely, i.e. all motes transition to **Operational** when the join trace is on. You may want to repeat this experiment a few times to get a feel for the variability.

Having 5 motes means, on average, that someone will hear the advertisement faster than one mote would in a single mote network so you may see the first mote join quicker in the 5-mote case. However, if too many motes synch up at once they will contend for the same shared Access Point (AP) links which can slow down the overall network join time.

Return the manager to the "fast" downstream frame before proceeding:

```
> set network bandwidthProfile=P1
```

- Reset the manager and log back in.

3.2.5 Measuring Time to Recover from a Lost Mote

One of the reasons a mesh is important is the robustness to path failures. In star and tree structures, a single RF path can represent a single point of failure for one or many devices data. In evaluating the mesh, many observers will want to see the mesh recover from a path failure. The most reliable way to make a path fail is to power off a device. Many observers also care a great deal about recovery time after the loss of a device, so this test can address both.

First we must decide what we mean by 'recovery'. If a user walks up to a 10 device mesh, and powers down one device, we consider the network recovered if the only change in data delivery is that the one node powered down stops sending data. If all other nodes continue to send data at their configured rates without interruption, then we consider the recovery time to be zero. The network survived the loss of a node with no disturbance to the remainder of the network. Another way to look at it is to establish some metric for quality of service, like data latency, that is being met prior to powering down a node, and then looking for the loss of the node to cause that QoS metric to degrade, and look for that QoS metric to get back to where it was before. This involves more of a judgement call from the observer and depending on the reporting rates might not be easy to assign a hard time value. The third way to identify recovery is to power down a node that is a parent of one or more nodes, and measure the time it takes for the network to detect that a parent has been lost and establish a new parent.

To summarize the three different "time to recover" test motivations are:

1. Uninterrupted data delivery. If powering down a mote causes no interruption to data delivery from any other motes, then time to recover = 0. If data delivery is interrupted, then time to recover is the time until data delivery from all nodes its restored
2. QoS data delivery. Baseline the QoS of a network through data analysis. Power down a node, and note degradation in the QoS metric for some or all nodes. Time to recover is the time until that previous QoS metric is met again
3. Time to repair the mesh. Look at the mesh. Power down a node that will cause other nodes to have only a single parent. Time to recover is the time it takes for the network to detect the loss and re-establish a full mesh

Test 1 and 2 are essentially the same test. We recommend you deploy a network, connect to CLI and turn on the stats trace:

```
> trace latency on
```

This will show a line for every packet that is received from motes. That line will include the Mote ID and the latency of that packet. Capture this text for some time, and then power down a mote. The mote you power down could be chosen randomly or you could specifically identify a mote with many children in the mesh. Note an approximate timestamp in your text capture when you powered down a node. Let the network continue to run for several minutes. Then you should be able to plot the data from this text capture to identify the moment when the last packet was received from the powered down mote, and after that time you should be able to identify if there are any gaps or delays in data delivery from any other nodes. Test 3 involves a different trace function on the manager. Deploy your initial network, and observe that a full mesh has been established (i.e. if numparents=2, then all motes will have two parents except one mote will have exactly one parent). Start a text capture of the manager io trace:

```
> trace io on
```

Select and power down a mote. Note the approximate timestamp for the moment you powered the mote down and watch the events associated with detecting paths that have failed and the activity associated with establishing new paths. After a few minutes stop the trace, and convince yourself that a full mesh has been restored. The 'time to recover' is the time from when you powered down the mote to the timestamp of the final 'Link Add' event in your capture. Expected results

1. If the network was a good mesh to start with, we expect no data loss and no additional mote loss due to the removal of a single device.
2. At moderate report rates (one packet per 5s or slower), we expect no disturbance in QoS. At faster report rates, you may see latency delays associated with a single device loss. Motes that had ~200 ms latency might have ~500 ms latency for a short time. Recovery time should be between one and two minutes.
3. Powering down a mote should be detected and repaired between one and two minutes

3.3 Latency

3.3.1 Measuring Latency

Activating the manager CLI statistics trace provides a latency measure for every upstream packet received at the manager:

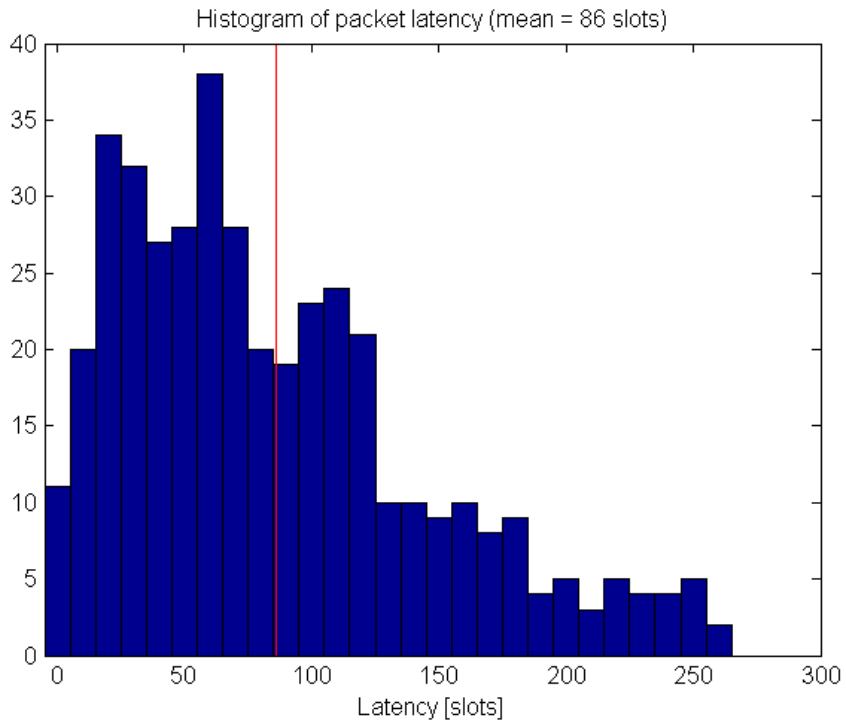
```
> trace latency on

Mote 2 Latency:
  stamp:    13:31:44.548
  received: 13:31:45.191
  latency:  00:00:00.643

Mote 2 Hops: 1 Average hops:
```

This trace has three lines of report for each upstream packet. The first line shows when he packet was sent by the mote. The second line shows when the packet was received at the manager. The third line is the difference between these two counts, here as 643 ms. The time between the sensor generating the data, and the notification at the manager will be slightly longer, as there may be some queueing delay. The first line also shows how many hops upstream the packet took. This trace also shows the hops for this mote.

You can collect a large number of latency trace prints and plot a distribution. It should look something like this (this is for IP, in WirelessHART it would be 4x larger since the upstream frame is 4x longer):



On average, unless path stability is very low, we expect a per-hop upstream latency of < 1/2 the upstream superframe length.

3.3.2 Comparison with a Star Topology

In ZigBee networks, sensor nodes typically report to powered, always-on routers. As a result, data latency can be very low - until the link fails.

Dust's networks offer a number of means to achieve low data latency - through a pipe which affects a single mote and its ancestors, through network-wide base bandwidth settings, or through per-mote services.

Try the following experiment:

- Place the motes within a few meters of each other and form a mesh network.
- Measure average latency as described in the previous section. By default each mote will send temperature data once every 30 s, so you should take > 10 minutes of data.
- Form a star network, by forcing the motes to be non-routing. This is done via the manager CLI on a per-mote basis. E.g. to set mote 2 to be non-routing:

```
> set mote 2 enableRouting=false
```

This command only works after a mote has joined at least once (so it has a short address that the manager remembers) - you will need to reset the mote and have it rejoin for it to become non-routing.

- Measure average latency - you should see it has decreased slightly, as now motes will only have the AP as a parent. This "star" configuration makes motes vulnerable to link failures, and didn't really improve latency much, and isn't really the right solution to improve latency.
- Set the motes back to routing-capable, and turn the upstream pipe on to a single mote - this is done by the `exec activateFastPipe` manager CLI command:

```
> exec activateFastPipe 00-17-0D-00-00-30-00-70 up
activateFastPipe command has been accepted.
```

This command uses EUI-64 addresses. You can get the mapping between EUI-64 and Mote ID using the `sm` manager CLI command:

```
> sm
Current time: 07/03/12 13:58:06 ASN: 405387
Elapsed time: 0 days, 01:07:41
      MAC                 MoteId  Age Jn      UpTime   Fr Nbrs Links State
00-17-0D-00-00-19-2D-B2   ap  1        1      01:07:33  6    1    25 Oper
00-17-0D-00-00-30-00-70       2     9  2      00:31:34  2    1    10 Oper
```

- Measure the average latency - it should be dramatically lower, despite motes occasionally routing through peers.
- Turn off the pipe:

```
> exec deactivateFastPipe
deactivateFastPipe command has been accepted.
```

- Increase the base bandwidth in the network. The default is 100000 ms. This is done with the `set network` manager CLI command:

```
> set network requestedBasePkPeriod=1000
```

- After this, a manager reset is required.
- After reconnecting to the manager and joining all motes, turn CLI latency trace back on
- Measure average latency - it should have dropped. Even though motes are not publishing any faster, they have received more links, so latency decreases.
- Set the base bandwidth back to 100000 ms.

```
> set network requestedBasePkPeriod=100000
```

- Request a service on one mote.
- Measure average latency - you should see that it dropped for the mote with a service, and may have dropped a little on other motes that have that mote as a parent.

3.3.3 Tradeoff Between Power and Latency

In general there is a tradeoff between power and latency. The more links a mote has, the lower the latency will be on any given packet, since we tend to space links relatively evenly throughout the superframe. This means that motes that forward traffic have lower latencies than motes that don't, even if they generate packets at the same rate. It also means that a mote can improve latency by asking for services at a mean latency target even if the data generation rate is lower.

Some of the power measurement experiments in the Power section show the correlation between power and latency.

3.3.4 Roundtrip Latency

Dust's networks are designed to primarily act as data collection networks, so most bandwidth is spent on upstream traffic. While the use of services can improve the upstream contribution, if fast (< 2 s per message) call-response is needed, a bidirectional pipe can be used. Try the following experiment:

- Ping a mote You should see a print similar to the following on the manager CLI (this is using profile P2):

```
> ping 2
> [14:16:05] Ping mote 2: reply #1: 20.528s 1 hops [24.0C 3.645V]
[14:16:05] Ping mote 2: sent 1, rcvd 1, 0% lost. Ave.roundtrip: 20.528s hops: 1
```

Repeat this several times and measure the roundtrip latency (delta between TX time and RX time, here 1382 ms)

- Turn on bidirectional pipe:

```
> exec activateFastPipe 00-17-0D-00-00-30-00-70 bidir
activateFastPipe command has been accepted.
```

- Repeat the ping test of step one - you should see a dramatic improvement in roundtrip latency.

```
> ping 2
> [14:18:09] Ping mote 2: reply #1: 0.357s 1 hops [24.0C 3.652V]
[14:18:09] Ping mote 2: sent 1, rcvd 1, 0% lost. Ave.roundtrip: 0.357s hops: 1
```

3.4 Channel Hopping and Range

Unlike most other 15.4 based systems, Dust's products employ a Time Slotted Channel Hopping MAC - this means that transmissions are spread out over all (or some - see Blacklisting below) channels in the 2.4 GHz band. This has the advantage that the system's performance depends on the average channel stability, rather than a single channel's stability. The following test will show you the how to measure single channel stability and see the effect of channel hopping. This test requires that you have access to a 802.11g Wi-Fi router and can set it on a particular channel - set the router to 802.11 channel 6 - this should cover mote channels 4–7. You will also need to configure your manager and a mote for radio test.

Note that in the various APIs below, channels are numbered 0-15. These correspond to channels 11-26 in the IEEE channel numbering scheme.

3.4.1 Using radiotest for Single Channel Measurements

This description places the manager in the role of transmitter, and the mote in the role of receiver, but the test can be run easily with roles swapped.

- At the manager Linux prompt, log into the Access Point CLI to invoke radiotest commands. You can get to the Linux prompt if you are in nwconsole by using the `logout` command.

```
dust@manager:~$ /opt/dust-manager/bin/apdconsole.py
Conect to 127.0.0.1:55551
>
```

- Configure it to be the transmitter, setting it for a packet test, on channel 0, 1000 packets - wait to hit return until you've configured the mote as receiver. Login is optional in radiotest mode.

```
> radiotest tx 0 1000
```

- On the mote CLI, place into radio test mode and reset. Then configure it to receive on channel 0 for two minutes

```
> radiotest on
> reset
HART Mote 1.0.0-104

> radiotest rx 0 120
```

- At this point hit return on the manager to start the test. After the manager is done sending, record the number of packets received on the mote CLI:

```
> radiotest stat
Radio Test Statistics
OkCnt : 998
FailCnt : 2
```

The ratio of received to sent is the path stability (or packet success ratio) for this channel. Note that 1000 - *OkCnt* may not equal *FailCnt*, since the manager only counts packet that it can lock onto.

Repeat for all channels. You should see a dip in path stability around the channels being used by the Wi-Fi router - note how the dip isn't as big as you might expect. This is in part due to the fact that the router is also duty cycling, and is only on a small fraction of the time.

3.4.2 Range Testing

The radiotest commands can also be used to do range testing. Range is the distance at which two devices can reliably exchange packets, but it is also a grey area, since the following all affect the ability of two devices to communicate at a distance:

- Reflectors - things the radio signal bounces off of, including the ground.
- Obstacles - things the radio signal must go through
- Presence of interferers
- Antenna design
- Radio power settings

The simplest test is to repeat the single channel measurement described above, and repeat for all channels, since a real network will use all channels. You will find that the packet error rate is strongly influenced by the height of the radio off the ground, due to self-interference between the line-of-sight path and the bounce path. With motes at 1 m elevation, you may see a drop off in packet success ratio near 50 m spacing, only to have it improve farther out. Motes placed several meters above an open field should be able to communicate at hundreds of meters.

After this test, return the mote to normal operation:

```
> radiotest off
OK
> reset
```

And exit the apd console on manager - you will need to log back into the nwconsole CLI as described at the beginning of this document.

```
> logout
apdconsole exit!
dust@manager:~$
```

⚠ The DC9003 boards come with an integrated chip antenna that has significantly less gain (-2.3 dBi average vs +2 dBi) than a dipole antenna. Range measurements should take this difference into account.

3.4.3 Blacklisting

The manager provides an API to blacklist certain channels in the case of a known interferer. In general, unless you know that there is a strong interferer (such as a very busy Wi-Fi router) you should not blacklist

- Form a 5 mote network - after all motes have joined, run for an hour. Use the manager CLI to look at the path stability for the network as a whole. At no time should you see packets lost due to the interferer.

```
> show stat short 1
It is now .................. 07/03/12 14:47:25.
This interval started at ... 07/03/12 14:30:00.
There are 8 valid 15 minute intervals stored.
NOTE: you may be expecting packets that are still in transit!
--------------------------------NETWORK STATS---------------------------------
PkArr  PkLost  PkTx(Fail/ Mic/ Seq)  PkRx  Relia.  Latency   Stability
    7       0    --(   -/   0/   0)    --    100%   1.23s         --
--------------------------------MOTE STATS-----------------------------------
 Id Rx    Lost  Tx    Rx    Fwd   Drop  Dup   Ltncy Jn Hop avQ mxQ me ne Chg  T
  2     7     0     6     3     0     0     0  1.23  0   1   0   2  0  0 108 24
```

- Blacklist channels 4-7, and reset the manager. The blacklist persists through reset.

```
> set blacklist frequency=2425 frequency=2430 frequency=2435 frequency=2440
```

- After the network has formed, run for another hour, and observe the path stability - it should have improved.

Return the manager to 15 channels before proceeding:

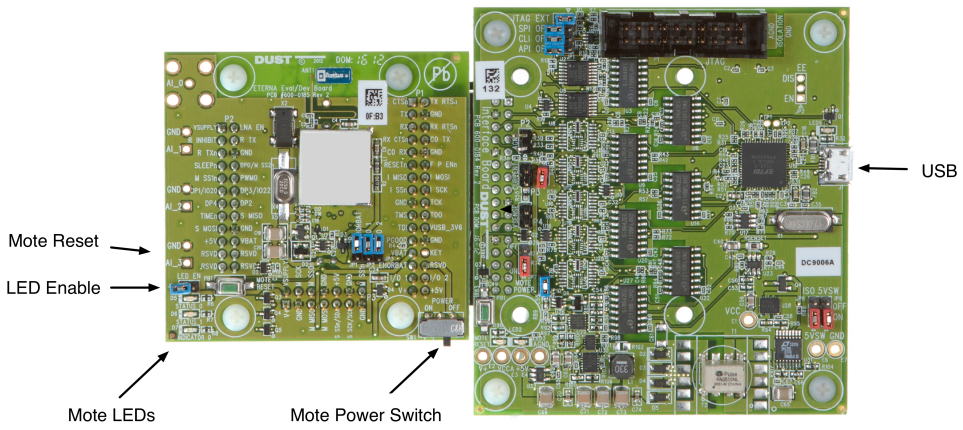
```
> set blacklist frequency=2480
```

3.5 Power

How to measure power on an eval/dev board:

In order to measure mote current, you should use a DC9003A-C/DC9006 combination. You can either connect a scope across the jumper marked VSENSE on the DC9006 - this measures the voltage drop across a 10 Ω series resistor, or you can remove both the CURRENT jumper (black) and the red jumper behind it, and connect an averaging current meter across the CURRENT jumper pins.

> ⚠ The LED_EN jumper on the DC9003A-Cboard must be disabled or the current measurement will be incorrect.



Suggested Measurements:

- Idle current - this is when a mote has been reset, but no join API command has been issued. The mote must be configured in **slave** mode to prevent automatic joining. After conducting idle current tests, return the mote to master mode
- Searching - Measure at 5% duty cycle, and 100% duty cycle. You should see the current change from ~250 µA average to 5 mA average.
- Mote joined - After joining, the mote current should be about 30 µA. This current will be cut in half once the manager has slowed the advertising rate it should be around 14 µA. You can verify the advertising rate by typing `show mote x` in the CLI where x is the Mote ID of your mote. The advertisement interval is 1.28 seconds for the higher-power fast advertising and 20 seconds for the lower-power slow advertising.

Dust provides a SmartMesh Power and Performance Estimator to help estimate mote average current under various use cases. Compare your test results to the spreadsheet results.

3.6 Mesh Behavior

Dust's networks automatically form meshes - each mote has multiple neighbors through which it can send or forward data. This setting is a manager policy that is visible with the manager CLI command >show config (see above). The config element that sets this policy is numparents, and as you can see above the default value is 2. The manager will make sure every mote has at least the number of parents indicated by *frameup.parents*, provided there are no loops in the mesh. Try to sketch this on your white board and you'll find that in a one mote network, that one mote will have only the AP mote as a parent. Add a second mote, and one mote will get two parents, but the other will be left with only one parent, to avoid making a loop. Regardless how many motes you add, there will always be one 'single parent mote'.

3.6.1 Testing the Mesh

The built in Admin Toolset allows you to visualize the mesh as it forms. Having two parents is the best balance between robustness and power consumption under most operating conditions. Robustness is achieved by having multiple parents. To confirm that we recommend you find a mote in your network that is a parent to one or many motes. Power that mote off. Data delivery from that mote will stop, but the children will continue to send data. Use 'trace stats on' as above to show a live stream of data from all the children motes, which are sending data through their other parents. You may see a temporary increase in latency for a few minutes, and if you capture that trace to a file you can show this graphically. The manager will take a few minutes to reassign parents and links in the system to account for the loss of the mote you powered down.

3.6.2 Changing Mesh Policies

The parameter *frameup.parents* can be given any of 4 values: 1, 2, 3 or 4. If you set numparents to 1, your network will be a tree, not a mesh. Each mote will have one parent, and that one parent may not be the AP. This is better than a star in that a tree supports multi-hopping. But like a star, a tree network has numerous 'single points of failure'. If you power down the parent of a mote in a tree, the child mote will drop off the network because it has no connections to any other devices. It will reset and will have to join again, most likely resulting in data loss. A tree structure can be slightly lower power, because each device only has to listen to one parent for downstream messages and only has to maintain synchronization with one parent. Furthermore it is assumed that data latency will be more bounded in a tree, because individual packets can't 'wander' around the mesh. We believe that the likelihood of network collapse and data loss far outweigh the potential benefit in power and latency.

Two is the default and has a nice balance of robustness vs power. Setting *frameup.parents* to 3 or even 4 increases the 'meshiness' of the system and gives the network additional robustness at the cost of somewhat higher power. We recommend increasing the number of parents in networks where you know that paths will be breaking frequently, for example if some motes are moving or periodic interferers are present. See the Application note Identifying and Mitigating the Effects of Interference.

3.7 Data Rates

The SmartMesh SDK contains a sample application called PkGen that allows you to set the rate of packet publication on one or more motes in a SmartMesh WirelessHART network. With it you can answer the following questions:

- What does real data flow in a network look like?
- How fast can a single mote go?
- How fast can all motes go? With upstream backbone on?

Remember final data rates depend upon the number of motes, the services each mote is asking for (or base bandwidth in addition to or instead of services), topology, and path stability, but this tool gives a good sample of the kind of flexibility you can expect.

4 Application Note: Data Publishing for SmartMesh WirelessHART

This note describes the specific steps required for an OEM microprocessor's firmware application to connect to a mote in the network and then have it send data over the wireless network. The key concepts from the SmartMesh WirelessHART User's Guide and SmartMesh WirelessHART Mote API Guide are brought together here. The User's Guide describes the details of how the mote and the OEM microprocessor interact and combine to form a system.

4.1 Request and Response Packet Formatting

All packets must use HDLC Encapsulation as described in the SmartMesh WirelessHART Mote API Guide. This means that the packet is preceded and followed by a framing byte 0x7E. A 2-byte frame check sequence (FCS) bytes must be computed for the packet and appended before the trailing framing byte. Instances of 0x7D and 0x7E in the payload or FCS require escaping, as described in the Mote API Guide in the "Protocol" section under "Packet Format."

Flag	Payload	FCS	Flag
7E	Packet Payload	2 bytes	7E

4.1.1 Header

The payload of both response and request packets begins with a 3-byte header. The first byte indicates the type of command or notification being sent or responded to. The next byte is the length of the remaining payload (not counting header or response code). The third byte is a set of flags. A complete description of the header and flags can be found in the Protocol section of the mote API guide under "Packet Format."

Request

Header	Payload
3 bytes	Request Payload

Response

Header	Response Code	Payload
3 bytes	1 byte Response	Response Payload

⚠ The length byte does not include the 3-byte header, or the response code.

4.1.2 Flag Byte

The third byte in the three byte header is the flag byte. The flag byte currently has only 3 usable bits:

Bit 0 - Request/Response: Bit 0 is cleared (0) if the packet is a request and is set (1) if the packet is a response. Acknowledgement (ACK) packets are response packets and must have this bit set.

Bit 1 - Packet ID: For every new request packet by the OEM micro, the packet ID bit must be toggled.

Bit 3 - SYNC bit setting rule: The SYNC bit must be set (1) on the first request packet OEM micro sends to the mote. It must be cleared (0) for subsequent requests. The SYNC bit is set (1) on boot event packet, which is the first request packet sent from the mote to the OEM micro.

The OEM micro, when acknowledging a packet, should set bit 0, echo the packet ID bit from the received packet, and echo the SYNC bit from the received packet. All other bits should be 0.

4.2 Basic Steps

There are a total of 6 basic steps that the OEM micro needs to perform in order to get the mote to join a network and start publishing data. The setup of the OEM micro's serial port is a very important first step in order for the rest of the communication to be successful. To generate sample firmware for a simple publish application, we need only to code these in order to get the combined micro+mote system to start publishing data. The steps that follow are:

1. Set up the serial interface
2. ACK the mote boot event
3. Perform pre-join mote configuration
4. Issue *join* command
5. Monitor join progress
6. Publish data

1. Set up the Serial Interface: The communication between the OEM micro and the mote takes place on the API serial port using a 4-wire protocol (UART Mode 4: TX, RX, UART_TX_CTSn, UART_TX_RTSn lines) by default. By default, the API port settings are: baud rate is 115.2 Kbps, 8 data bits, no parity, 1 stop bit.

⚠ The baud rate on the LTC5800-WHM can be changed to 9600 if required, by updating the fuse table programming on the mote.

2. ACK the Mote Boot Event: Whenever the mote is powered up (or reset) it sends a boot event packet on the API port. The boot event packet is as follows:

```
7E 0F 09 08 00 00 00 01 01 00 00 00 00 D7 67 7E
```

The mote will continue to send this packet until it is explicitly acknowledged by the OEM micro. If the serial port settings are correct then the micro should be able to see this packet show up in its receive buffer. If the micro's serial port is configured incorrectly, one can see this packet on a scope or Logic Analyzer on the Tx pin of the mote. It takes 6-7 seconds after power up to receive this packet.

The ACK packet for this event is:

```
7E 0F 00 05 00 9F 30 7E
```

The OEM micro needs to respond with this packet in order for the mote to stop sending the boot event packet. Responding with this ACK will move the mote from state 0 (**Init**) to state 1 (**Idle**).

3. Perform Pre-Join Mote Configuration: Once the mote boot event has been acknowledged and it is in the **Idle** state, it is ready to join the network. Before issuing the *join* command, you may want to change one or more of the pre-join configuration settings. These settings may be left as the factory default at first, but should be later adjusted for optimal operation. Some of these configurable parameters are:

- *setParameter* <*networkId*> - The default Network ID is 0x04CD (1229).
- *setParameter* <*joinKey*>: The default join key is 0x 445553544E4554574F524B53524F434B. For the highest level of security, each mote should have a unique join key.
- *setParameter* <*joinDutyCycle*>: Default is 0xFF or 100% (of 255)

See the Mote API Guide - the "Commands" section details each API, and the "Definitions" section gives the encoding for arguments.

Assuming it had been set previously to a different value, the packet to change the *joinDutyCycle* parameter to 0xFF or 100% (255) is:

```
7E 01 02 04 06 FF 8C C5 7E
```

Header = 01 02 04, so this is the *setParameter* command (0x01), length of payload = 2 bytes (0x02), and the flags byte has bit 2 set. If this is the first packet from OEM micro to mote, the sync flag should also be set (not shown).

Payload = 06 FF, so this is the *joinDutyCycle* being changed (0x06), and the value is 100%, i.e. 255 (0xFF)

The mote will then reply with this response:

```
7E 01 01 05 00 06 xx xx 7E
```

Header = 01 01 09, so this is the *setParameter* command (0x01), length of payload = 1 bytes (0x01), and the flags byte has bits 3 and 0 set.

Response code = 00, so the command had no errors (0x00)

Payload = 06 , so this is the *joinDutyCycle* being changed (0x06)

Calculation of the correct checksum is left to the reader.

4. Issue *join* Command: Now that the pre-join configuration is done, the OEM micro is ready to issue a *join* command to the mote that will prompt it to enter the process of joining a network (specified by its Network ID). The join packet is:

```
7E 06 00 04 31 56 7E
```

The mote will respond with an acknowledgement:

```
7E 06 00 05 00 FC C9 7E
```

5. Monitor Join Process: In the process of joining the mote will send several event notifications to the OEM micro that must be acknowledged. These notifications are as follows:

mote *operational* notification:

```
7E 0F 09 02 00 00 00 20 05 00 00 00 00 EB FA 7E
```

OEM micro acknowledgement:

```
7E 0F 00 05 00 9F 30 7E
```

mote *svcChange* notification (a mote will receive a maintenance service from the manager to allow it to respond to messages):

```
7E 0E 0C 00 00 00 80 01 03 02 F9 81 00 01 86 A0 A3 D8 7E
```

OEM micro acknowledgement:

```
7E 0E 00 05 00 24 2C 7E
```

mote *timeChanged* notification:

```
7E 0F 09 02 00 00 00 04 04 00 00 00 00 63 64 7E
```

OEM micro acknowledgement:

```
7E 0F 00 05 00 9F 30 7E
```

One may optionally query the mote to get its status by sending the following packet- “get mote status” packet (request by OEM micro):

```
7E 02 01 00 0E 0A 76 7E
```

6. Publish Data:

Once *operational*, the mote is available to send the data up the wireless network to the Gateway. In a HART compliant network, application commands are used to determine which kinds of sensor data (e.g. temperature, humidity, voltage, current) , or "variables" are available, and how often to publish or "burst" them.

SmartMesh networks are built with a certain amount of bandwidth - the *base bandwidth*. This bandwidth is assigned to a *maintenance* service, which is intended for responses to manager requests. A HART compliant mote must also request a *publish* service (see below) before publishing data. If all motes will publish below the base bandwidth (which is configurable), and HART compliance isn't required, the OEM micro need not request any services from the manager. The factory-default base bandwidth is sufficient for sensors to publish data every 100 seconds. If faster publish rates are required, the default settings on the manager can simply be increased so that it gives out a higher amount of bandwidth to each mote when it joins.

To build the actual packet to be sent on the serial port, one may use the **dust_hdlc.h** and **dust_hdlc.c** files from the SmartMesh SDK. The functions in these files will generate the proper CRC and append the 7E delimiters to the “payload” that is required to be sent on the serial port.

The Starter kit guide provides an example of sending a payload to the mote which might be a good place to start from. Try to use the exact same payload and see if the code can build the exact same packet:

Payload bytes(6) (to be sent from OEM micro to the mote):

```
AA BB CC DD EE FF
```

Resulting Send data packet:

```
05 0F 04 F9 81 80 02 02 00 00 FF 06 AA BB CC DD EE FF
```

Final HDLC packet (from OEM micro to the mote):

```
7E 05 0F 04 F9 81 80 02 02 00 00 FF 06 AA BB CC DD EE FF 7E
```

ACK (response received from the Mote):

```
7E 05 00 01 00 51 8B 7E
```

The Manager CLI can be used to verify that the packet, sent by the mote, was successfully received by the manager. Refer to the SmartMesh WirelessHART Manager CLI Guide for details.

4.3 Helpful Hints

Ignore Packet ID

Ignore “Packet ID” when sending commands. See the flag fields in the command structure. Set bit 2 in all commands sent. If the “do not ignore” bit is set then the “packet id” field needs to toggle for every command sent. A command with a wrong packet id value will “hang” …… so one may think the command was sent when it actually was NOT.

4.4 Next Steps

Now that you have successfully sent a data payload to the manager you can look at the mote API guide for more commands and detailed descriptions of customization options.

- Test Radio Commands - these can be used for manufacturing tests to verify the top level assembly, e.g. that the antenna has been properly connected.
  The test radio commands (*testRadioTx* and *testRadioRx*) are described in detail in the .SmartMesh WirelessHART Mote API Guide.
- Services
  Once *operational*, the mote is available to send the data through the wireless network. If the application calls for publishing data at different rates for different devices, this is a *heterogeneous bandwidth* network and all devices should request services. We recommend that OEM integrators always use services, since they are simple to implement and offer the most flexibility. Services are discussed the "Services" section of the SmartMesh WirelessHART User's Guide, and "Bandwidth and Latency" section of the SmartMesh WirelessHART User's Guide.

5 Application Note: Using WireShark to Troubleshoot HART Manager Connections

If the network is running properly and you can interact with the manager through the CLI but are having difficulty with the XML interface, you can use WireShark to help in the troubleshooting process. Such XML interface issues include: not being able to interact at all, missing blocks of data, or having intermittent failures. This is not a perfect troubleshooting application for our needs, but it does provide some help.

5.1 Download and Install

WireShark can be downloaded at http://www.wireshark.org/download.html. There are installers available for multiple platforms.

If installing on Windows, the installer may ask install to WinPCap, and this should be accepted. Configure WinPCap to be started at boot (default).

Wireshark requires that the admin account have read/write access to the files /dev/bpf* in order to do packet captures on OS X. By default only root has rw access. This can be changed on a temporary basis in Terminal. Open terminal, and type:

```
sudo chmod 777 /dev/bpf*
```

You will be prompted for your administrator password. This change will persist until reboot - since this is potentially a security risk, we recommend rebooting or changing the privileges back after using wireshark.

5.2 External Documentation

How to setup a capture

5.3 Using the Software

5.3.1 Capturing Traffic

The following settings allow better viewing of packets:

- To recognize HTTP on the control channel, add Control Channel port number (e.g. 4445) to **Edit->Preferences->Protocols->HTTP->TCP Ports**
- To recognize XML contents of notifications, set **Edit->Preferences->Protocols->XML->Use Heuristics for TCP**

Capture traffic on the PC interface that has visibility to manager packets. Note that if you are trying to capture traffic between the manager and another computer, you must be on the same LAN segment. An Ethernet hub or switch in the middle will usually keep the traffic separated on different segments. For better performance, you may want to disable live update of the windows with **Capture Options->Update** list of packets in real time.

5.3.2 Inspecting Data

Once you have captured the packets of interest, you can apply filters to display only relevant data.

All packets with the manager

```
ip.addr==<ipaddr_val>
```

5.3.3 Control Channel Packets

Control channel is XML-RPC traffic that runs on top of http. HTTP runs on top of TCP.

Use the following filter expression to display packets on the control channel

```
ip.addr==<ipaddr_val> and tcp.port == <port>
```

E.g., if the Manager has IP address 192.168.1.10 and control channel is on port 4445(default), the expression is:

```
ip.addr==192.168.1.10 and tcp.port == 4445
```

5.3.4 Notification Channel Packets

Use the following filter expression to display packets on the notification channel

```
ip.addr==<ipaddr_val> and tcp.port == <port>
```

E.g., if the Manager has IP address 192.168.1.10 and notification channel is on port 24112(default), the expression is:

```
ip.addr==192.168.1.10 and tcp.port == 24112
```

6 Application Note: Testing for WirelessHART Certification

6.1 Building a WirelessHART Mote

A device must meet a number of requirements to become a compliant WirelessHART device.

At a minimum, it must:

- Support all Universal Commands (HCF Spec-127)
- Support some Common Practice Commands (HCF Spec-151), particularly those regarding burst control (publishing), namely Commands 33, 103, 104, 105, 107, 108, and 109
- Receive a Manufacturer Identification Code and Extended Device Code from the HART Communication Foundation (requires membership). The Extended Device Code is needed to format a valid HART MAC address
- Fully implement the Dust APIs to support *setNVParameter<macAddress>*, *setParameter<hartDeviceStatus>* and *setParameter<hartDeviceInfo>*, and command termination of Gateway originated commands intended for the sensor processor
- Fully implement services and application domains
- Have a wired HART modem
- Undergo certification testing

6.2 Working with the WirelessHART Test System

The HART Communication Foundation (HCF) sells two kits – HCF-Kit 192 (a Linux based system for testing wired HART protocol) and HCF-Kit 193 (the wireless test hardware and test scripts), and a multi-channel packet sniffer called WiAnalys. Each new device requires that the vendor have an HCF issued Vendor ID, and that HCF issue a 2-Byte extended device ID (EID) for that device. The certification process starts with the Vendor subjecting the device to a series of tests, and submitting the resulting logs to the HCF. HCF then reviews the tests, and issues a compliance report that states whether the device passes or fails, and why.

Tests consist of running a C++ test script (e.g. `tmlxxx.cpp`) on the Linux system, watching that the test completes successfully (a log is generated called `tmlxxx_testlog`), and capturing the wireless packets exchanged. The WiAnalys system allows the user to select an output tab delimited text file for the wireless packet capture.

It is very important that you run your test in a quiet (RF) environment – the test system picks up packets from other running systems – this seems to lead to a lot of test errors that have nothing to do with the device under test (DUT).

6.2.1 How to Analyze Test Results

Assuming your device can pass the TML100 – TML102 series tests, much of your SmartMesh WirelessHART mote API code has been implemented correctly and that the device can join a network. If not, you should return to the Mote Serial API documentation – the HART test system is not appropriate for general debugging of the mote API.

- Review the TML Test Descriptions document – it explains what the test is trying to do, and will help you focus on what to look for.
- Review the tmlxxx_testlog file – look for the word “Error” in the file and note the timestamp, then look in the export file for corresponding issues.
- Match request to response - note any packets that are not responded to. One issue observed is that the test system increases the security counter without the DUT having responded, and subsequent requests are ignored. This is not a DUT error - it is a test error. Verify that request transport makes sense - only reliable requests are answered. Non-wireless commands are passed to your processor for transport termination - make sure that you have implemented this interface correctly if you see problems with non-wireless commands.

Since the kit-192 system and the wiAnalys system are not on the same computer, there may be a delta between the timestamps in the testlog and the wiAnalys export file. To match events between the two logs, look for the join request line in the testlog:2012:2:23:17:40:12::---- Info :: JoinDevice :: Join request Received

and compare that with the ”Date and Time” column entry corresponding to the join request in the export file:

DATE AND TIME	PAYLOAD
2012-02-23 17:41:25.890	[ReadUniqueIdent cmd=0, bc=23]*[ReadLongTag cmd=20, bc=33]*[ReportNbrSignalLevel cmd=787, bc=7]

6.2.2 How to Read an Export File

Each test consists of a series of requests from the test system, followed by a response from the DUT. The export file from WiAnalys contains many tab delimited columns of data. Some of these are important, some not, depending upon what problem you are trying to identify.

Columns that are always important:

- Packet number – useful reference when reporting a particular issue, e.g. “the problem starts at packet 1234
- Date & Time – Useful for comparing testlog and export file
- PDU – Color coding rows by PDU type and To/From columns helps to visualize the packet exchange
- To – MAC layer recipient. A successfully received packet should be followed by an ACK packet back to the sender on the next line. This helps clarify whether a failure to respond to a command was because the command was incorrectly formatted, or the DUT simply didn’t hear it.
- From – MAC layer sender
- Dest – Net layer recipient. F980 is manager, F981 is gateway.

- Src – Net layer sender
- PayloadHex – Raw bytes. Useful for pairing requests with responses. Also contains error code – format for responses is command, length, errorcode, payload, e.g. [965, 6, 0, 01 00 66 01 06] – Write Superframe command, 6 bytes, RC=0 (no error), and the payload, which can be parsed using HCF Spec 155 as a reference.
- Payload – Parsed output. Useful in seeing what the state of the DUT is when you don't want to parse the raw payload.
- SLCnt – Security layer counter. Useful for understanding why the DUT may not respond to a command that it ACKed on the MAC layer. If the counter is out of sequence, the mote will not respond.

The following python script can be used to remove unwanted columns, advertisements and bad packets from the trace, making them easier to read. In addition to the columns listed above, it preserves the priority, graph ID, source route, security layer control, and transport control and counters, which can also be of use for particular tests. It also preserves WiAnalys messages that tell when the test advances to a new step.

```
#!/usr/bin/python
import sys
#expects a tab delimited input file
if len(sys.argv) != 2:
    print "Usage: Untitled.py filename"
    sys.exit(1)
else:
    #open an output file with "parsed" appended to name and CSV
    input = open(sys.argv[1],"r")
    a = sys.argv[1].partition('.')[0]
    output = open(a+'_parsed.txt', "w")

output.write("Packet\tTime\tPDU\tPriority\tTo\tFrom\tGID\tDest\tSrc\tSR1\tSR2\tHex\tPayload\tSLCTL\tSL
for line in input:
        temp = [temp2.strip() for temp2 in line.split('\t')]
        if temp[6] != '0x0000':
            print 'skip bad'
        elif temp[6] == 'Packet Status':
            print 'skip first line'
        elif temp[9] == 'Adv':
            print 'skip adv'
        else:
            output.write(temp[2] + '\t' + temp[3] + '\t' + temp[9] + '\t' +\
            temp[10] + '\t' +temp[14] + '\t' +temp[15] + '\t' +\
            temp[29] + '\t' +temp[30] + '\t' +temp[31] + '\t' +\
            temp[33] + '\t' +temp[34] + '\t' +temp[35] + '\t' +\
            temp[36] + '\t' +temp[37] + '\t' +temp[38] + '\t' +\
            temp[44] +'\n')
 output.close()
 sys.exit(0)
```

7 Application Note: Custom Configuring a Manager for Application Specific Performance

7.1 Example Application

SmartMesh WirelessHART managers were designed to operate safely under a wide range of conditions and network topologies. Sometimes this means that performance for a particular application is not optimal. Specifically, the manager is designed to handle 250-500 motes, all of which could be at 1-hop, and form low-power, data reporting networks with little downstream traffic.

A customer wants to use the manager for bulk configuration of mote properties. The following limitations are stated:

- Network only needs to handle 64 motes at a time
- Configuration speed is the primary metric - data reliability is not a key metric, as there will be no reporting
- Network will be formed with motes in close proximity to the manager, so the network is expected to be flat, dense, and with good path stability.
- No configuration changes should be required on the motes

In this case, the default downstream superframe size is placing a limit on the number of motes that can join and be configured simultaneously. The solution is to enable manual superframe settings and configure the downstream frame to be much shorter. A shorter superframe repeats more often, so a shorter downstream superframe means that there is more downstream bandwidth available into the network. This will allow the application to send packets faster and bulk configure its devices in a shorter interval.

A shorter superframe also means more receive links for the motes. As such, this configuration increases power requirements at the motes. This tradeoff explains why we don't use a shorter superframe as a default setting, even for smaller networks.

7.2 Manager Configuration

Using the manager CLI, the `set network` command is used. First the current settings are noted:

```
> get network
netName:                myNet
networkId:              1229
optimizationEnable:     true
maxMotes:               251
numMotes:               1
accessPointPA:          true
ccaEnabled:             false
requestedBasePkPeriod:  100000
minServicesPkPeriod:    400
minPipePkPeriod:        480
bandwidthProfile:       P1
manualUSFrameSize:      1024
manualDSFrameSize:      256
manualAdvFrameSize:     128
netQueueSize:           0
userQueueSize:          0
locationMode:           off
```

Next, set the bandwidth profile to `Manual` - this makes manual settings take effect next time the manager boots. The manager prints network settings after each change (not shown).

```
> set network bandwidthProfile=Manual
```

The downstream frame size is set to 32 - this makes the downstream schedule 8x faster (as it was 256 by default), but limits the number of 1-hop motes. Since all motes are close together, and upstream data is minimal, this is not a concern.

```
> set network manualDSFrameSize=32
```

Finally, the base bandwidth is set to 5000 to give motes a few more upstream links to respond more quickly to commands.

```
> set network requestedBasePkPeriod=5000
```

At this point the manager is rebooted from the Linux shell or power cycled.

7.3 Verify New Settings

Logging back into the manager and CLI allows us to confirm that the settings were used properly:

```
> get network
netName:                myNet
networkId:              1229
optimizationEnable:     true
maxMotes:               251
numMotes:               1
accessPointPA:          true
ccaEnabled:             false
requestedBasePkPeriod:  5000
minServicesPkPeriod:    400
minPipePkPeriod:        480
bandwidthProfile:       Manual
manualUSFrameSize:      1024
manualDSFrameSize:      32
manualAdvFrameSize:     128
netQueueSize:           0
userQueueSize:          0
locationMode:           off
```

If the *bandwidthProfile* is set back to P1, the manual frame sizes will be overridden after the next boot event but the *requestedBasePkPeriod* will remain at 5000.

8 Application Note: Manager Linux Shell Commands

This document describes various Linux shell scripts and commands which can be used to emulate most of the functionality found in the SmartMesh WirelessHART Admin Toolset user interface and other key features.

8.1 Logging In

Users or a user-written program can access these scripts by logging in via an SSH connection (or terminal program) under the `dust` username. The factory default password is `dust`. To change the dust account password (e.g. changing it to `newPassword!`), enter the following at the user prompt ($):

```
$ sudo set-dust-password newPassword!
```

8.2 Root Password

The Linux root password is described in user documentation and should be considered public knowledge. However, users cannot log in remotely as root. Users must first login as `dust`, and then change users to root via the Linux `su` command. For that reason it is critical that if the manager is addressable from the outside world (i.e. it is not behind a firewall), then the `dust` password be changed. Changing the `dust` password is sufficient for most attack scenarios. The additional access one gets as `root` does not add significant risk, although it does allow the user to delete some critical files. If the attack scenario is that someone who has `dust` access needs to be stopped from doing things as `root`, then changing the root password will mitigate that. The steps are:

- login as `dust`
- switch to `root` (note that the character before the '3' is a lowercase letter 'l'):

  ```
  $ su sur+cycl3s
  ```

- change the root password (e.g. to `newRootPassword!`)

  ```
  $ set-root-password newRootPassword!
  ```

8.3 Setting IP Address

To change manager to a DHCP-assigned IP address:

```
$ sudo ifswitch-to-dhcp
```

To set manager to a static IP, note that the gateway (router) address and netmask are optional fields:

```
$ sudo ifswitch-to-static IP-ADDRESS [GATEWAY] [NETMASK]
```

8.4 Restart/Reboot

The following commands will stop your SmartMesh Manager software and restart the entire network:

```
$ sudo /etc/init.d/dust-manager stop
$ sudo /etc/init.d/dust-manager start
```

The following command will reboot your SmartMesh Manager hardware. This will restart the manager software and force your network to reform.

```
$ sudo reboot
```

If connected via SSH, your connection will be dropped. When reboot is complete (after a few minutes), you will need to re-establish your SSH connection.

8.5 Clear All Motes

The following commands can be used to clear all information stored about your motes. As motes rejoin the network, this information will be stored again. Note that this procedure requires temporarily stopping the manager software process, deleting the mote database, then restarting the manager process.

```
$ sudo /etc/init.d/dust-manager stop
$ sudo rm -f /root/motes.xml
$ sudo /etc/init.d/dust-manager start
```

8.6 Restore Factory Settings

The following command scripts will restore manager settings to the factory defaults. Manager will need to be rebooted after the execution of these commands, as described earlier.

```
$ sudo restore-factory-conf
```

⚠ Restoring to factory configuration will reset Ethernet back to using a static IP address. If you wish to retain a DHCP address, use the *restore-dcc-conf* script instead.

8.7 Software Update Using I-Packages

The following commands can be used to upload and install software updates (contained in .ipk files) to the manager.

Step 1 - Copy the .ipk file to `/tmp` or `/home/dust` on the manager

This is typically done with a program on your computer that supports the Linux `scp` command, such as PSCP (part of PuTTY). You will need to know the IP address of your manager, and provide the `dust` user login credentials. It is also possible to execute `scp` from the manager, but your computer must allow incoming SSH connections.

Step 2 - After logging in to the manager, execute the `ipkg` install command with the full path to the ipkg. For example, to install `example.ipk` in `/tmp`, you would type:

```
$ sudo ipkg install /tmp/example.ipk
```

⚠ The ipackage install script will prompt you about replacing several files - if you wish to return to factory settings, you should opt to overwrite files. The default setting is to keep the existing files.

8.8 Setting Time

The `set-time` command can be used for manually setting the real-time clock, or to synchronize the manager real-time clock to an NTP server. The general syntax is as follows:

```
$ sudo set-time { [-z <timeZone>] [{-t <time>} | {-s <server1> <server2> <server3>}] [-q]}
```

- **timeZone**: name of timezone to use (see Valid Time Zone values at the end of this document)

- **time**: current time in format MMDDhhmm[[CC][YY][.ss]]
- **serverX**: names of ntp timeservers to use
- -q: instead of prompts, use old values

8.8.1 Examples

To set the real time clock to pacific time, January 1st, 12:34 AM:

```
$ sudo set-time -z PST -t 01011234
```

To change the time server:

```
$ sudo set-time -z PST -s 0.us.pool.ntp.org
```

Normally time changes are made before the network is started. For time changes to take affect in a running network, the manager must be restarted:

```
$ sudo /etc/init.d/dust-manager restart
```

8.9 Serial Port Settings to PPP

By default, the Manager is configured to allow console login on the serial port labeled Serial 1. It is also possible to configure Serial 1 to support PPP access. In this mode, Serial 1 is always expecting a connection from a PPP client. The following commands will establish serial settings on the Serial 1 port for PPP.

```
$ sudo config-login disable
$ sudo config-ppp enable
```

8.9.1 PPP Configuration

The default PPP configuration expects a client to use the following serial port settings:

- 115200 baud
- 8 data bits, 1 stop bit, no parity (8N1)
- no flow control

When connected, the default PPP configuration expects the Manager IP address to be 192.168.101.10 and the PPP client IP address to be 192.168.101.11.

8.9.2 Windows Client configuration for PPP

The built-in Windows PPP connection expects an extra handshake from the PPP server. To incorporate with Windows, the PPP connection must be configured to expect a Windows client, use:

```
$ sudo config-ppp enable-windows
```

8.9.3 Linux Client configuration for PPP

Different Linux distributions will vary in their PPP setups. In general you will need to

1. Install the PPP package (if one is not included in your distribution)
2. Configure the PPP options (e.g. in `/etc/ppp/options.ttyS1`)
3. Launch a PPP client or configure the PPP daemon (pppd)

8.10 Data Logging

The datalog utility is used for capturing upstream packets from the motes to the Manager. The data logging feature requires external storage for the capture file; storage is provided by an SD (or MMC) card. The Linux command line utility datalog is available in the LTP5903-xxx Manager with SW revision 4.0.2 or higher.

8.10.1 Starting and Stopping a Capture Session – "datalog"

The data capture can be started from the Linux command prompt by typing `datalog`. This will display the usage information:

```
Usage: /opt/dust-manager/bin/datalog start <id> [-h <header>] [-t]
       /opt/dust-manager/bin/datalog stop <id>
       /opt/dust-manager/bin/datalog status [<id>]
    start  - begin a capture session
       id          - id of capture session
       -h <header> - use the given header file
       -t          - include packet timestamp with each data record
    stop   - stop a capture session
    status - print a description of capture sessions
```

For the `start` command, the *id* is an integer. The *id* is appended to the name of the capture file. By default, the capture file is always prefixed by "capture", so the name of the capture file will be `capture.id`. By default, the capture file is created on the SD card, i.e. in the `/media/card` directory mounted on `/dev/mmcblk0p1`. If no card is mounted then datalog will return an error. When a capture is started, all other files are removed from the capture directory.

A *header file* is an arbitrary file that is kept alongside the capture log, i.e. the header file is not removed along with older capture files. The header file is assumed to be located in the capture directory. It can be used to identify the contents of the capture file, e.g.:

```
echo "Running experiment XYZ" > /media/card/my-header-file
datalog start 101 -h my-header-file -t
```

8.10.2 Configuring a Capture

The configuration parameters such as the prefix, the directory, etc. can be changed in the datalog configuration file, `/opt/dust-manager/conf/settings/datalog.conf`.

8.10.3 Reading a Capture – "datalogConvert"

A capture file is a binary file containing timestamped upstream data packets. A capture file can be read as text using the datalogConvert utility:

```
datalogConvert <inputFile> <outputFile>
```

- *inputFile* - file to convert
- *outputFile* - file with ASCII representation

The result will be stored in the *outputFile* and will be in a text format:

[timestamp,] mac address, hex data

The **outputFile** will contain a timestamp for each line if the datalog capture was started with the "-t" parameter.

8.11 Valid Time Zone Values

EST (US/Eastern)	CST (US/Central)	MST (US/Mountain)	PST (US/Pacific)
Africa/Cairo	Africa/Casablanca	Africa/Harare	Africa/Nairobi
America/Mexico_City	America/La_Paz	America/Lima	America/Buenos_Aires

America/Sao_Paulo	Asia/Calcutta	Asia/Almaty	Asia/Baku
Asia/Bangkok	Asia/Colombo	Asia/Kabul	Asia/Karachi
Asia/Kuwait	Asia/Magadan	Asia/Muscat	Asia/Yekaterinburg
Asia/Vladivostok	Asia/Yakutsk	Asia/Tokyo	Asia/Tehran
Asia/Taipei	Asia/Seoul	Asia/Istanbul	Asia/Hong_Kong
Atlantic/Azores	Australia/Perth	Australia/Hobart	Australia/Darwin
Australia/Brisbane	Australia/Adelaide	Australia/Canberra	Canada/Newfoundland
Canada/Saskatchewan	Canada/Atlantic	Etc/GMT-2	Europe/Vienna
Europe/Helsinki	Europe/Budapest	Europe/Amsterdam	Europe/Bucharest
Europe/Moscow	Europe/Belgrade	Greenwich	Israel
Kwajalein	Pacific/Fiji	Pacific/Guam	Pacific/Auckland
Pacific/Samoa	Singapore	US/Hawaii	US/Arizona
US/Eastern	US/Pacific	US/East-Indiana	US/Mountain
US/Central	US/Alaska		

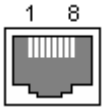
9 Application Note: How to Interact Programmatically with the Manager Serial 1 Port

This document describes how to write a simple program to read and write data to the SmartMesh WirelessHART Manager serial port Serial 1. This port is labeled Serial 1 on the LTP5903CEN-WHR, and is available on J10 on the LTP5903-WHR.

9.1 Hardware Specifications

Connecting to the Manager's Serial 1 port requires an adapter to convert from the RJ45 connector to DB9. See "Assembling a 9-pin D-SUB Adapter for Serial 1" in the SmartMesh WirelessHART User's Guide for information on assembling an RJ45 to 9-pin D-SUB adapter.

Port	Description	Signaling
Serial 1	UART 5-pin	RS232 levels



RJ45 Connector

Pin	Signal Description
1	TX out of manager
2	RTS out of manager
3	RX into manager
4	GND
5	GND
6	CTS into manager
7	Not connected
8	GND

RJ45 Connector Pinout

9.2 Example Program

A simple Serial Echo example program, `serial_echo.py`, is used to communicate with the serial port is written in Python – Python was chosen for its readability and interactivity.

Python 2.6.1 is included on the SmartMesh WirelessHART Manager platform. This embedded version of python does not contain the complete python standard library, but it does contain many useful modules. The example program uses the Pyserial module to provide the low-level serial I/O. The Pyserial module is not part of the python standard library.

9.2.1 Code Walkthrough

There are three interesting sections in this example:

1. Opening the serial port
2. The loop that reads from the serial port
3. The loop that writes to the serial port

The serial port is opened by importing the serial module and creating a Serial object. The configuration of the serial port takes place by assigning values to members, then the port is opened.

```
def main(port = DEFAULT_PORT, speed = DEFAULT_BAUDRATE):
    # connect to serial port
    serial_fh = serial.Serial()
    serial_fh.port     = port
    serial_fh.baudrate = speed
    serial_fh.rtscts   = False
    serial_fh.timeout  = 1     # required so that the reader thread can exit
    serial_fh.open()
    ...
```

The loop that reads from the serial port is very simple. The select function waits for the serial port to be readable. When input is available, it reads a single character and prints it to `stdout`.

```
def main(...):
    ...
    while True:
        result = select.select([serial_fh], [], [], 1.0)
        if serial_fh in result[0]:
            serial_data = serial_fh.read()
            print 'Read: ' + serial_data
            #sys.stdout.write(serial_data)
```

The loop that writes to the serial port is also very simple. This loop runs as a thread so that it's independent of the loop that's reading from the serial port. It uses the raw_input function to read a line of input from the user and writes that data to the serial port.

```
class StdinThread(threading.Thread):
    def __init__(self, serial_fh):
        threading.Thread.__init__(self)
        self.daemon = True
        self.serial_fh = serial_fh

    def run(self):
        print 'Starting input loop'
        while True:
            try:
                in_data = raw_input('> ')
                self.serial_fh.write(in_data)
                self.serial_fh.flush()
            except EOFError:
                print 'Ignoring EOF'
                pass


def main(...):
    ...
    input_thread = StdinThread(serial_fh)
    input_thread.start()
    ...
```

9.2.2 Setup Instructions

1. Disable the login program that runs on the serial port by default

```
dust@manager$ sudo config-login disable
```

2. Fix permissions on the serial device (as root):

```
root@manager# chmod 666 /dev/ttyS1
```

3. Copy the Serial Echo software to the manager. Set up the pyserial package:

```
dust@manager$ cd serial-echo-example
dust@manager$ tar zxf pyserial-2.6.tar.gz
dust@manager$ cd pyserial-2.6
dust@manager$ python setup.py build
dust@manager$ export PYTHONPATH=`pwd`/build/lib
dust@manager$ cd ..
```

4. Open a serial terminal on the connected computer.
5. Run `serial_echo.py`

```
dust@manager$ cd serial-echo-example
dust@manager$ python serial_echo.py
```

9.2.3 Cleanup

Depending on how the Serial Echo program terminates, it could leave the console in a state where user input is not echoed. The easiest way to clear this state is to log out and log back in.

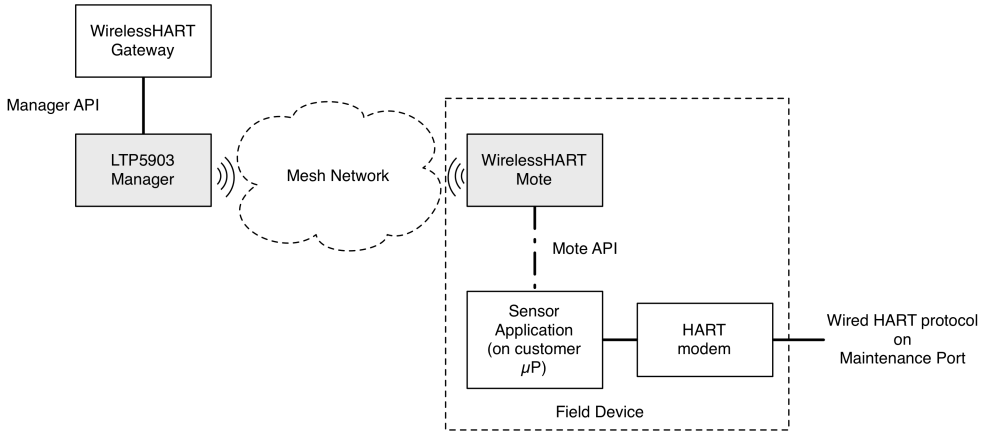
9.3 References

- Python documentation
- Pyserial module, Pyserial documentation
- serial_echo.py source

10 Application Note: Building a WirelessHART Compliant Device

10.1 Introduction

The SmartMesh software (the "network stack") on an WirelessHART mote (LTC5800-WHM, LTP590x-WHM) is a subset of a WirelessHART device. A *field device* consists of the network stack component running on the WirelessHART mote , communicating via a serial API to a sensor application running on a customer microcontroller (or in the future, the WirelessHART mote itself). The system is illustrated in the following figure:



10.1.1 Joining a WirelessHART Network

To join a WirelessHART network, the sensor application only needs to do a few things:

1. Set up the serial interface (baud rate, data bits, stop bits)
2. Acknowledge the mote boot event
3. Perform pre-join mote configuration as needed - e.g. select a join duty cycle
4. Issue the *join* command
5. Monitor join progress
6. Once operational, request services (required in a WirelessHART compliant device)
7. Publish data

Once in the network, the sensor application may see commands coming from the WirelessHART Gateway - the sensor application will be expected to respond to these commands. Gateways may also reject a device not presenting a proper HART address. Linear does not make Gateway software, so the user should contact vendors whose Gateways they wish to support to understand what is required to function properly.

10.1.2 Base Requirements for WirelessHART Compliance

For a field device to be WirelessHART compliant, there are additional requirements. The WirelessHART mote terminates commands in the wireless command space (768-1023, 64,512-64,765). The sensor processor application is responsible for terminating all commands outside of the wireless command space (those outside 768-1023 and 64,512-64,765). Any command intended for, but not supported by the sensor application should return an RC 64 (Command not Implemented). A compliant device needs a 2-Byte Manufacturer Identification Code and 2-Byte Expanded Device Type (EDT) from the HART Communication Foundation (HCF). The EDT is needed to form a valid HART MAC address.

At a minimum, a WirelessHART compliant device must:

- Support all Universal Commands (HCF Spec-127)
- Support some Common Practice Commands (HCF Spec-151) , particularly those regarding burst control (publishing).
- Implement the following serial APIs: *join*, *send*, *hartPayload*, *disconnect*, *setNVParameter<macAddress>*, *setParameter<hartDeviceStatus, setParameter<hartDeviceInfo>*, *dataReceived, serviceIndication,* and *events.*
- Properly support command termination of Gateway originated commands intended for the sensor processor (universal, common practice, and vendor specific). There are several WirelessHART Gateway vendors, and they may behave differently, i.e. some may use additional common practice commands than those listed here. It is the integrators responsibility to test against gateways they wish to support. For detailed examples of packet formats, see the application note "Data Publishing for SmartMesh WirelessHART"
- Fully implement services and application domains
- Implement configuration change counters per HCF specifications
- Pass wireless compliance testing. A device must also have a wired HART modem that it exposes on a "maintenance port" in order to use the HCF test system.
- Pass wired device testing

The application is also responsible for forwarding wireless commands received over the HART modem interface, in some cases using direct Dust APIs, in others using the *hartPayload*() API.

Universal Command Specification (HCF Spec-127)

Commands that require working code to support WirelessHART testing are indicated with an * in the list below, however all Universal commands are required in order to support wired testing. Universal commands can come to the application via one of two interfaces: the serial "maintenance" port, or wirelessly, where they will come through the *dataReceived* API. Responses to commands must go back on the interface where they arrived. Commands affecting the configuration changed counter are indicated with a (CC).

- **Command 0 - Read Unique Identifier***
  - Note: The mote sends this as part of the join request after receiving the contents from a *setParameter<hartDeviceInfo>* command, however in general the application must be able to respond to a command 0 request on the wired or wireless interface.
- **Command 1 - Read Primary Variable***
- **Command 2 - Read Loop Current And Percent Of Range***
- **Command 3 - Read Dynamic Variables And Loop Current***
- **Command 6 - Write Polling Address***
- **Command 7 - Read Loop Configuration**
- **Command 8 - Read Dynamic Variable Classifications**
- **Command 9 - Read Device Variables with Status***
- **Command 11 - Read Unique Identifier Associated With Tag**
- **Command 12 - Read Message***
- **Command 13 - Read Tag, Descriptor, Date**
- **Command 14 - Read Primary Variable Transducer Information**
- **Command 15 - Read Device Information***
- **Command 16 - Read Final Assembly Number**
- **Command 17 - Write Message**
- **Command 18 - Write Tag, Descriptor, Date**
- **Command 19 - Write Final Assembly Number**
- **Command 20 - Read Long Tag***
- **Command 21 - Read Unique Identifier Associated With Long Tag***
- **Command 22 - Write Long Tag***
  - Note: When received, the application should call *setParameter<hartDeviceInfo>* with the new tag.
- **Command 31 – Escape for 2-byte commands***
  - Note: This is not a real command – it is used to send 2-byte commands (e.g. spec-155) to the device over the wired interface. The 1st 2 bytes of the data field contain the command. It must be acknowledged on the wired port, then the 2-byte command should be passed to the network stack via the *hartPayload* API.
- **Command 38 - Reset Configuration Changed Flag**
- **Command 48 - Read Additional Device Status***

Common Practice Command Specification (HCF Spec-151)

The following Common practice commands are required for WirelessHART testing. Details on each command are found in spec-151. Common practice commands can come to the application via one of two interfaces: the serial port, or wirelessly, where they will come through the *dataReceived* API. Responses to commands must go back on the interface where they arrived. Changes to burst mode must persist through reset. Commands affecting the configuration changed counter are indicated with a (CC).

Of the following commands, 109 and 117 result in the application making a service request to the network stack. Others read or write configuration relative to what is being published. The purpose of the listed common practice commands is to configure publishing in the sensor application - the network stack does not comprehend publish parameters, rather it reacts to the application using the *requestService* and *send* APIs.

- **Command 42 - Perform Device Reset**
- **Command 54 - Read Device Variable Information**
- **Command 102 - Map Sub-device to Burst Message (CC)**
- **Command 103 - Write Burst Period (CC)**
- **Command 104 - Write Burst Trigger (CC)**
- **Command 105 - Read Burst Mode Configuration**
- **Command 107 - Write Burst Device Variables (CC)**
- **Command 108 - Write Burst Mode Command Number (CC)**
- **Command 109 - Burst Mode Control (CC)**
- **Command 111 - Transfer Service Control (CC)**
- **Command 116 - Write Event Notification Bit Mask (CC)**
- **Command 117 - Write Event Notification Timing (CC)**
- **Command 118 - Event Notification Control (CC)**

Serial API commands

The sensor application communicates with the WirelessHART mote using a serial Application Programming Interface (API). It consists of synchronous commands to configure the device, cause it to search for a network, send data, etc., and asynchronous notifications for received data, mote state transitions, service changes, etc.

See the SmartMesh WirelessHART User's Guide for explanation of networking concepts, and the SmartMesh WirelessHART Mote API Guide for details on the syntax of API commands and the HDLC encoding used to frame messages.

Configuration Change Counter

The application must maintain a 16-bit configuration changed (CC) counter. This counter is incremented each time a command affecting the CC counter is successfully received. It is nonvolatile. Device must maintain 1 bit in the counter for each master (GW or Manager) - see spec-127.
In addition to the Universal and Common Practice commands that affect the configuration changed counter, the following Wireless (Spec-155) commands also affect the CC counter and apply to field devices:

- 771 Force Join Mode
- 773 Write Network Id
- 775 Write Network Tag
- 815 Add Device List Table Entry (optional – not implemented)
- 816 Delete Device List Table Entry (optional – not implemented)

Refer to the Mote section in the SmartMesh WirelessHART User's Guide for implementation details.

11 Application Note: Using the Powered Backbone to Improve Latency

11.1 Introduction

All SmartMesh managers have a variety of configuration *knobs* that can be used to adjust key network performance metrics such as average mote power and message latency. There may be several different ways to tune a network to solve a particular problem – this document presents the *Powered Backbone* (or *backbone*) feature in SmartMesh WirelessHART and details when it should and should not be used in networks. Enabling the backbone adds a superframe for low-latency communication upstream. This comes without additional energy costs at battery-powered devices.

Limitations:

- In the current version of the SmartMesh WirelessHART products, there is no way to have a downstream backbone.
- The backbone size is chosen before booting up the manager and cannot be changed without a manager reset.
- The backbone size in SmartMesh WirelessHART is limited to 4, 8, or 16 slots.
- Power source settings presented at mote join determine backbone behavior - changing power source settings after join will not affect the backbone behavior once assigned.

Example Power calculations in this document are done with values from the SmartMesh Power and Performance Estimator.

11.1.1 General Motivation

The backbone was developed to achieve ZigBee-like low-latency upstream. In ZigBee networks, the routers (full function devices) are powered and all other nodes (reduced function devices) have to be within range of a router. Low latency comes at the cost of a) routers burning 10's of mA in receive continuously, and b) risk of catastrophic collapse in the case of a large amount of collisions. In a SmartMesh WirelessHART network with the backboneEnabled parameter set to "true", any mote can transmit upstream in a backbone slot provided it has a parent that is a set to line-powered, and furthermore, all motes still have dedicated upstream transmit links so that there is no possibility of a network collapse.

We use the power source setting on the mote to determine its behavior in the backbone. Motes with a "Line" source, if they themselves are not transmitting in a backbone slot, will listen for packets from their children. Powered devices have a lot of idle RX events, and these do use energy because the radio is in in receive mode to be ready if a packet does come in. Motes with other source values do not have receive links in the backbone, so they can still be low-power devices. They will have a lot of idle TX events, but these do not cost any energy when using Eterna motes.The backboneSize field can be set to values of 4, 8, and 16 slots, indicating the length of the backbone superframe. Lower numbers indicate more links per second.

All backbone activity happens on shared links. In parallel, the rest of the network continues to operate on higher-priority dedicated links. The backbone links are on a different channel offset than all other activity so even though traffic on the backbone might collide with other backbone traffic, it does not impact the reliable communication that still occurs on the dedicated links. Under no circumstances will the backbone decrease the overall reliability or increase the latency of a network.

11.1.2 Limitations for DN2510 Motes

Motes from the DN2510 product family cannot participate as routers for the backbone superframe due to software limitations. The manager will only assign TX links in the backbone for DN2510 motes, however, they can still be routers in the regular upstream superframe. The manager makes these decisions automatically; the sensor processor on the DN2510 isn't required to do anything special to participate in the backbone. Unlike Eterna motes, DN2510 motes do use energy for idle TX events. The charge consumed each time the mote has a TX link that goes unused is 4 µC, so, for example, a DN2510 mote participating in a 4-slot backbone can be expected to burn about 100 µA more than one without the backbone. In mixed DN2510 and Eterna networks, it can thus be advantageous to run with the longest backbone superframe that meets the latency requirements.

Though all applications will function with mixed DN2510/Eterna networks, unless otherwise specified, the remainder of this Application Note assumes that the network consists solely of Eterna motes.

11.1.3 Settings to Enable RX in the Backbone

SmartMesh WirelessHART motes report their power source in their join request packet. The sensor processor can use the *setNVParameter<powerInfo>* command to set the power source field. The manager uses the "Line" power source (0x00), as the flag for enabling a mote to have RX links in the backbone. Settings of "Battery" (0x01) and "Rechargeable" (0x02) mean the mote will only get TX links in the backbone. This highest setting must be provided by the sensor application and should only be used if the mote truly is powered or has a very large battery supply, since motes with receive links in the backbone can draw > 500 µA.

11.2 Application: Low-latency Alarms

Network

- Backbone superframe length *backboneSize*: 8

Expected Average Performance Metrics

- Latency: about 160 ms per hop
- Additional power at leaf nodes: none
- Additional power at routing devices: 190 µA

For a low-latency alarm network, we need to make sure that every mote is in range of a line-powered mote. The line-powered motes then form the backbone and any mote is at most one hop away from this backbone. As such, every mote can transmit in any slot. If the transmission fails, the transmitting mote assumes that the failure was due to a packet collision since the backbone slots are shared. The mote has a random exponentially increasing backoff to correctly use this contention-based set of links. There is nothing preventing the backbone from extending multiple hops from the AP. Taking average stability and the backoff mechanism into account, the mean latency per hop will be about 160 ms, or sixteen slots. However, because the backbone can have collisions, the application must be able to tolerate the latency delivered by the parallel dedicated links in the worst case.

For routing purposes, the dedicated upstream and backbone links are treated equally. If a mote has an upstream packet and a dedicated upstream link to its parent in the next slot, it will transmit the packet on the dedicated link rather than the shared backbone link, and the parent will also listen on the dedicated link. This will necessarily be on a different channel offset so another child of the same parent trying to transmit on the backbone link will not collide with the first transmission. Similarly, if a mote has any other link, such as a join listen link or a downstream RX link, it will service this instead of servicing the upstream backbone link. Also, a packet going multiple hops could end up traveling some hops on dedicated links and some on the backbone links. All of this will happen as a random result of which link is available first for the mote and if any backoff occurs.

A mote will only have one backbone parent at a time, this will be one of the (usually two) upstream parents. The backbone links are not used to determine path failure, but if a path failure is detected on the dedicated links to the backbone parent, the mote will automatically start using the backbone to the second parent. The dedicated links were being used all along to this second parent, but the application again needs to be tolerant of longer-latency periods during path failures.

The upstream backbone is most enabling for a network that does not send much data but needs low latency when it does generate packets. For a 100-mote network with an average latency requirement below 500 ms, there just aren't enough slots at the AP to provide this with only dedicated links. Depending on the latency and power requirements, the other *backboneSize* settings of 4, or 16 slots can be used. The available backbone lengths in SmartMesh WirelessHART are perfect for applications with strict requirements on the tails of the latency distribution or difficulty with out-of-order packets; since all motes in the network have frequent opportunities to send queued packets upstream, there is less chance that a packet will get "stuck" on a low-traffic router mote and slow it down relative to other packets sourced by the same mote.

11.3 Unsuitable Use of Backbone: High-Traffic Networks

Network

- 30 pkt/s total traffic to the Access Point
- 500 motes

Expected Average Performance Metrics

- Additional power at nodes: up to 23 µA

Remember that dedicated links take priority over the backbone links. If the network is busy, the Access Point will have dedicated links in most slots to receive the upstream traffic. Because the motes do not know the Access Point's full schedule, they will often transmit and fail on a backbone link which the Access Point is busy listening on a dedicated link. If there are several motes contending for the backbone bandwidth in this way, then even with backoff, most backbone transmissions will fail. If the mote is sending full-sized packets, all of these failures could result in spending up to 23 µA in extra transmit attempts. This may not be a huge penalty for a routing device, but it can be serious for a low-power device that is expected to use 30 µA total. This scenario may arise as a network grows - a network that starts out with a small number of motes reporting infrequently would see a huge latency improvement with the backbone, but as additional motes are added, the latency benefit to these original motes will decrease, and the backbone links that were originally free could now come at significant cost due as they repeatedly fail. Also note that the network would need to be reset in order to deactivate the backbone in such a scenario.

Backbone failures count as packet failures, so path stability values reported by motes in an overused backbone network will be very low. Watch for stability values below 50% to help diagnose this issue. These affected paths will be most apparent when drawing the RSSI-stability waterfall curve.

12 Application Note: Monitoring SmartMesh WirelessHART Network Health

The SmartMesh WirelessHART managers provide a full set of mote, path, and network statistics over the CLI. However, it is sometimes desirable for the application to be able to monitor network health in real-time as motes report their current state. Starting with the 4.1.0 manager, this information is provided to the application in the form of notifications to which the client application can subscribe. A combination of monitoring these notifications and periodically querying the Mote and Mote Statistics over the API can enable an external application to provide network health and debugging feedback to the user. Some examples:

- By watching state changes, the application can tell if any motes are resetting.
- By watching neighbor health reports, the application can make sure that all motes have a sufficient number of good neighbors for recovery if paths fail.

This process is broken up into two parts. The first is the set of notifications that the application should be continuously monitoring upon their output from the manager. The second is a set of API calls that the application should make periodically to gather the rest of the required information. Ideally, the application is started at the same time as the network and monitors health for the entire lifetime of the network.

We recommend storing all notification data (perhaps breaking it into daily logs). If storage is a concern, then storing max, min, and FIR filtered average for each item (e.g. path x RSSI) may be sufficient for providing user feedback.

12.1 Health Reports

There are four types of Health Report (HR) sent by each mote in the network, and each is sent once every 15 minutes. Three of these are standard to all WirelessHART devices and one is a Dust vendor-specific HR. The application can ignore the information in the Neighbor Signal Levels HR (Command 787) as this information is more easily obtained by querying the Mote element over the API. The WirelessHART spec doesn't provide specific field names for the values reported in each HR, so we'll use descriptive names below and indicate the corresponding payload bytes where appropriate.

12.1.1 Neighbor Health List (Command 780)

This HR gives us information about all the used paths that the mote is involved in. From this HR, we want to capture: *neighborId* (Bytes 3-4), *neighborFlag* (Byte 5), *rssi* (Byte 6), *numTxPackets* (Bytes 7-8), *numTxFailures* (Bytes 9-10). Also note the time stamp on the notification.

For calculating stability of a path, we are interested in the HR from the child's perspective to calculate the path stability. The *neighborFlag* bitmap can be used to tell if a path is a child or parent path. In the case of a child path, the stability in percent is 100*(1-*numTxFailures*/*numTxPackets*). We want to record this stability along with the *rssi* of this path. We record these two items as a pair for each path.

12.1.2 Device HR (Command 779)

From this HR, we just want to record the maximum queue length the mote experienced over the last 15 minutes, *maxQ* (Byte 9).

12.1.3 Dust Device HR (Command 64515)

Here we want to pull out values that tell us how successful the mote was in receiving messages from the application. This will factor into our calculation of overall network availability later. From this HR, we want to capture: *numAppTxAttempts*, and *numAppTxFails*.

The availability of the mote, in percent, is 100*(1-*numAppTxFails*/*numAppTxAttempts*).

For calculating the overall network availability, define two new variables and initialize them to zero. Call them *netAppTxAttempts* and *netAppTxFails*.

To keep a running tally with each Dust Device HR:

- *netAppTxAttempts* += *numAppTxAttempts*
- *netAppTxFails* += *numAppTxFails*

12.2 Periodic Querying of Motes Element

We recommend polling the manager every 15 minutes with these API calls to match the period of the HR. The *getConfig* method can be used with the Motes element. See the WirelessHART Manager API document for more details.

Store: *macAddr, moteId, isAccessPoint, numJoins, goodNeighbors, numNeighbors, maxNumNeighbors, numLinks, maxNumLinks, linksPerSec, maxLinksPerSec*

12.3 Periodic Querying of Network Statistics Element

The manager keeps track of the network statistics in 15 minute intervals. The *getConfig* method can be used with the Network Statistics element. If called every 15 minutes, the "short" interval with index 0 should be used to get the most recent full interval. From this element, we only want to store the *netReliability* value.

12.4 Tests

These are the tests that can be run on all the collected data to provide warnings about network health.

The tests below list conditions that should raise warning flags, i.e. you want your network to **not** have any of these conditions.

12.4.1 Iterate Over All Motes

These tests are run on each mote individually.

Fewer Than 3 Good Neighbors

Every mote in the network should have three "good" neighbors, *i.e.* having a quality score over 50%. The manager keeps track of the count of good neighbors, so just verify that the manager reports at least 3 good neighbors for each mote in the *goodNeighbors* field.

Close to Memory and/or Power Limits

There are three things to check for each mote here, and each current value should be compared to the maximum value reported by the manager for each mote. Verify that:

- *maxNumNeighbors* - *numNeighbors* > 0
- *maxNumLinks* - *numLinks* > 5
- *maxLinksPerSec* - *linksPerSec* > 0

The manager can reserve some links for use with the pipe, backbone, and optimization; this is why we need a bit of a buffer on the link number. The limits check is not available for AP devices, i.e. those with *isAccessPoint* = true.

Recent Join

If the *numJoins* value has increased in the previous 15 minutes, it means the mote has reset and rejoined. Mote resets, especially if they happen in groups, are the biggest indication that there is something wrong with the network.

High Maximum Queue Size

There are random bursts of traffic in a network that are normal which can temporarily congest a mote, but we don't expect these events to come during every 15 minute interval. Any mote that reports a *maxQ* > 7 for two or more consecutive intervals should be flagged as having congestion issues.

12.4.2 Iterate Over All Paths

The HR notifications capture 15-minute snapshots of every path in the network. Iterating over these snapshots can help to identify bad paths.

Bad Stability-RSSI

Flag any path that has either:

- RSSI above -80 dBm and stability < 50%
- RSSI above -70 dBm and stability < 70%

These paths lie beneath our prototype RSSI-Stability waterfall curve. If there are occasional outliers here, it is not a large concern. Consistent points below either threshold for a mote can indicate a hardware problem or interference in the vicinity. See "Application Note: Identifying and Mitigating the Effects of Interference" for more details.

As we captured the time of the notification from each HR, it is easy to print out the times of the outlier points to see if they are temporally grouped.

12.4.3 Network Checks

These are properties that can be checked for the network as a whole.

Reliability < 99.9%

We can check the *netReliability* every 15 minutes to make sure the data is being delivered with at least "three nines" of reliability.

Availability < 99%

To calculate the average availability for the network every 15 minutes, use this formula: 100*(1-*netAppTxFails*/*netAppTxAttempts*).

12.5 Graphing

If the application is continuously running, each of the quantities above could be plotted with 15-minute resolution. We have previously seen daily rhythms in network stability and latency in networks deployed in industrial environments where the amount of interference and moving machinery varies greatly between the workday and nighttime. Often tracking these metrics over time is much more revealing than taking a single snapshot.

13 Application Note: Building Deep WirelessHART Networks

13.1 Introduction

Starting with version 4.1, the SmartMesh WirelessHART family is well suited to building networks spanning a long linear distance. This includes applications such as monitoring the environment in a mineshaft and transmission line monitoring, where sensors tend to be deployed in the same direction away from an egress point. Previously we recommended to limit SmartMesh WirelessHART networks to 8 hops. With a one-dimensional deployment in mind, packets from wireless sensor nodes ("motes") further from the manager will require more hops to get to their destination. We refer to these motes as being "deep" in the multihop network. There are some performance characteristics that are specific to such deep multihop networks, as compared to denser mesh networks:

1. The network will take longer to fully join.
2. The manager will be able to support lower total traffic limits. A maximum total egress of 8.2 packets per second should be respected.
3. More attention needs to be placed on deployment locations and connectivity.
4. Packet latency will increase with network depth.
5. The network should be limited to 100 motes.

13.2 Deployment Guidelines

As with all SmartMesh deployments, motes should all be deployed within range of three other potential parents to ensure network reliability. In the case of a linear deployment, this means all sensor motes must have three motes within range and closer to the manager. Furthermore, to preserve this property near the manager, we recommend placing some repeater motes (motes with or without sensors) close to the manager. If the radio range in the desired environment is *R*, then the deployment should be carried out according to Figure 1.

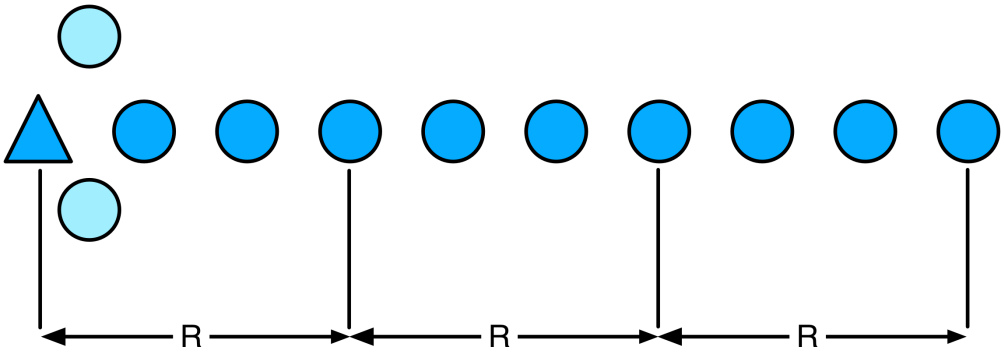


Figure 1. Each sensor mote (dark blue circle) has three upstream neighbors closer to the manager (triangle). The repeaters (light blue circles) provide traffic handling and spatial diversity.

If there are fewer sense points than this in the deployment, repeaters should be added to make up the required density of three potential parents per device. The total distance that can be covered is greatly affected by the deployment environment, which affects range. As drawn in Figure 1, if for example *R*=100m, a 100-mote network can monitor out to beyond 3 km. Placing the devices 20 to 30 meters off the ground and in line-of-sight can extend this range by 5x (i.e. > 15 km), and placing them in a mine shaft 1 meter off the ground can reduce it by half (1.5 km).

13.3 Determining Range

The two example deployments, mineshafts and transmission lines, are expected to lie at opposite ends of the device range spectrum due to the radio propagation characteristics in these very different environments. In either of these settings, there is no substitute for directly measuring device-to-device range with a pair of fully integrated motes using the actual finished or prototyped wireless sensor with the antenna you plan to use. This measured range informs both the number of repeaters and the maximum distance for the network. For this reason, we encourage OEMs to make range measurements as early as possible in the development cycle.

For more information on estimating range, see the application note "Planning a Deployment."

13.4 Mote and Manager Versions and Settings

To build deep networks, the user must have SmartMesh WirelessHART Manager version 4.1.1.7 or later and SmartMesh WirelessHART Mote version 1.1.1.76 or later. Deep networks must be built using only Eterna motes. If upgrading is necessary, it must be done before deployment.

In addition to the version requirement, the manager needs some configuration changes that are different from traditional deployments. These settings must also be changed before building the network and are persistent. The settings will make deep networks work better, but will make shallow networks less robust, so we recommend only making the changes when the network is expected to be more than 5 hops deep. Add the following lines to the dcc.ini file:

- `RELIABLE_MGR_UCAST_NUM_RETR=10`
- `TOP_FRAME_UP_LINKS=10` (refer to the calculation of L in the "Calculating Links" section)
- `TOP_NUM_DN_MCAST_LINKS=0`
- `TOP_LINK_OVRSUBSCR=2.0`
- `TOP_FRAME_ADV_APLST_BLD=2`
- `TOP_FRAME_ADV_APTX=2`

Then, from the manager CLI interface, type the following:

- `set network bandwidthProfile=Manual`
- `set network manualUSFrameSize=256`
- `set network numParents=3`

This needs to be followed by a manager reset.

No configuration changes are required at the motes.

13.5 Calculating Links

At a minimum, the number of links assigned should be set to 10 (as in the settings detailed above). More links may be required for faster reporting rates or lower latency in the network, and can be calculated using the following formula:

L = [2.3M/T]

Where the number of links is *L,* the number of motes including repeaters in the network is *M*, and the reporting interval is *T.*

The square bracket here indicates that you should round up. We recommend limiting the total network egress to 8.2 pkt/s, so to get the maximum throughput for 100 motes, this is one packet per 13 s per mote and a little extra to carry the health report packets. Calculating the link requirement in this case yields *L*=19. To reduce latency, at the cost of higher average energy use in the network, the value of *L* can be increased. At some point, depending on the number of 1-hop motes, the AP link table will fill up at around 200 total upstream receive links.

14 Application Note: Planning A Deployment

14.1 Estimating Range

Hardware integration choices influence how well devices can communicate with each other over a distance with antenna choice being the most obvious. Post-integration, device placement can change the effective range over orders of magnitude. At one end of the spectrum, devices placed on elevated poles or towers with clear line of sight to other motes in the network may have a range of 1000 m or more. At the other end, devices placed on the ground or next to large metal objects may have an effective range of 10 m or less. So when a customer asks you "What's the range of your radios", in some ways that is a meaningless or unanswerable question. You can refer to the datasheet for transmit power and receive sensitivity and the resulting link budget, but the customer will determine the range with choices they make in the development of their products and an evaluation in a real environment similar to their expected deployments.

We recommend that customers at the beginning of development plan on their devices working at a spacing of 50 m. Analysis of the first several 'typical' deployments can guide the typical range number up or down. Deployment planning simply requires that each mote, is being placed within this range of at least three existing devices. In order to form a reliable mesh, every device must have multiple neighbors and hence numerous opportunities to connect. Placing motes within range of only one other device along a maximally spaced string will result in a fragile network prone to mote resets and data loss.

14.2 Mapping out a Deployment

Once you have settled on a range for your environment, you can use a scale map to place motes at all the required sense points for the network. If possible, the AP should be located near the middle of the distribution of motes to reduce latency and mote power. Mark the AP location. Supposing that the range estimated above is 50 m, draw a circle with a 50 m radius around the manager. Not all motes within this circle will be able to communicate directly with the AP, but some motes outside the circle will, so on average it will balance out. The number of motes inside this circle approximates the number of 1-hop motes in the deployment.

Next draw a 100 m radius circle centered at the AP. The number of motes in the ring between 50 m and 100 m approximates the number of 2-hop motes. Repeat this process with circles of increasing radius until all motes have been encircled and note how many motes are in each hop. We'll use these hop counts in a minute.

There are two more things to check:

- Each mote, including the AP, should be within the estimated range of 3 other devices.
- The network should be no more than 8 hops. Deeper networks are indeed possible and should just work, but they are harder to model.

14.3 Estimating Power and Latency

Dust has provided a SmartMesh Power and Performance Estimator that estimates network performance for both product families. It can be used to estimate battery size for a given topology, packet rate, per hop latencies, and desired lifetime.

The inputs are:

- Number of motes at each hop
- Reporting interval and packet size
- Network configuration
- Path stability

With these inputs, the estimator first evaluates whether the network will form, and if so, gives:

- Mote average current by hop
- Mean latency by hop
- Network join time
- Joining mote current

We recommend using the example configurations provided to guide customer thinking on power budgeting.

There are many factors that influence power consumption and latency in actual deployments, including path stability (which changes over time) and the connectivity of the network as deployed (how many hops deep it is). The estimator provides reasonable average power for motes at a given hop - there may be motes with 3x the hop average. These estimates are for expectation setting - actual in-network performance will vary.

15 Application Note: Predicting Network Health

Evaluating the health of a deployed and running network is important to ensure that long-term performance targets are achievable. Network health verification is simple and based on interpreting readily available information. When a network fails this network health verification test, adding more motes in key locations will usually remedy the problem.

15.1 Motivation

Once a network has been deployed and is running, an important step is to verify that the network is healthy, and more importantly, that it will be healthy in the future. The network itself collects all the required diagnostic data which is then served up at the software interfaces of the manager. It is important that early on in the development of products that the developer integrating software to the manager knows that diagnostics are important. Building the tools to automate the health verification of networks is an extremely good investment that will instill confidence in the minds of end users, particularly those skeptical about wireless.

The process described below will let the user know if it is safe to walk away from a network (a "green light"), and also serves to identify the source of problems in the rare case that a properly installed network does have problems.

15.2 Overview

Verifying a network involves answering two multi-part questions:

1. Does the network LOOK GOOD?
2. Does the network have the building blocks to BE GOOD?

The desired answer to both these questions is YES. If the answer is YES to both these questions, the test has passed and the network in question should be expected to run well for the foreseeable future.

15.3 Does the Network LOOK GOOD?

This part of the network evaluation test involves answering three very simple observational questions about the network. They are:

Is the data reliability high? In any good deployment, the data delivery rate in the network be close to 100%. Dust networks are built with very few mechanisms for losing data, and data reliability of >99.99% is expected. Confirm that this is the case.

Is the joining behavior correct? The first part of this question is: did all your devices join? Only the installer can know for sure how many devices were deployed. They must make sure that if they put 100 devices out there, that 100 devices joined. The second part of this question is making sure that all devices joined precisely once. If a device joined more than once, has it been continuously live in the network long enough to make you confident it is not constantly dropping out and rejoining? A device that dropped out and rejoined once while the network was building is not as ominous as one that resets long after building is complete.

Does it look like a mesh? This question can be answered at a glance if there is a GUI for visualizing the network. If there is no GUI, then you simply check that all motes have two parents in the mesh. There should be exactly one mote in the one hop ring that has only one parent. That is OK and expected.

15.4 Does the Network Have the Building Blocks to BE GOOD?

This part of the network evaluation test involves answering three quantitative questions about the details of connectivity in the mesh. Those three questions are:

Are there enough motes in the one-hop ring? All traffic in the network converges at the AP mote, the mote connected to the network manager. That single mote is critical for all data to be delivered. By extension, all the *one-hop* motes that communicate directly with the AP are important as well. The hardest working motes in the mesh will be in the one-hop ring. Those are the motes that are forwarding the most traffic from their descendants. The more one-hop motes, the better for a network as there is more opportunity to balance the traffic and to survive a single mote reset. We never want to build a system where removing one mote will cause the loss of many motes' data. As a rule of thumb, there should be at least 5 motes or 10% of the total, whichever is larger, in the one-hop ring. If you have a 120 mote network with 10 one hop motes, it is not guaranteed that it will fall apart. But you should have good quantifiable guidelines here so a non-expert can answer a yes/no question, with actionable instructions on what to do when the answer is no.

Does every mote have enough good neighbors? This step involves waiting until 15 minutes after the last mote has joined, looking at all the discovered paths in the network, and making sure that every mote has enough good quality neighbors. The bare minimum is that every mote should have at least 3 good neighbors. A good neighbor is a neighbor that this mote can hear at >-75dBm or has >50% path stability. These paths do not have to be currently in use, they just have to be discovered and reported by the network.

Are any motes at or near their link limit? In all current products, motes at 90 links or more indicate a risk of bandwidth issues in the network.

16 Application Note: Common Problems and Solutions

16.1 Introduction

Networks are built with the goal of providing reliable services while keeping power as low as possible on the wireless devices. Since links use energy, motes are given as few links as possible to adequately carry the expected traffic through the mote during the joining and steady-state phases of the network lifetime. The manager depends on the motes accurately reporting their service requirements and each path averaging better than 50% stability. If there is a bottleneck, meaning that any mote has run out of links due to power or memory constraints, there may not be enough bandwidth to carry all the traffic from the descendants of this mote.

Symptoms of a low-performing network are:

- Slow formation time
- Mote resets
- Large variation in packet latency
- Manager reports lost packets from any mote

The causes of these performance issues are typically one or more of:

- Poor connectivity - motes do not have enough neighbors with good quality paths
- Interference - in-band WiFi or Bluetooth is present or a strong out-of-band interferer is nearby to lower path stability
- Oversubscription - motes are reporting more than their accepted service requests allow causing congestion

Motes report their internal and path statistics in Health Report packets. These statistics are broken up into 2 or 3 packets and are generated every 15 minutes. In particular, check the reported path stability values on the paths that are currently being used by the mote. The mote reports the maximum and average size of its internal packet queue. Dust networks are provisioned so as to rarely have more than one packet at any mote at any time, so a nonzero average queue length usually indicates a problem.

The manager also keeps track of the network topology and the link assignments at every mote in the network. This perspective can immediately identify if any motes have run into link limits or have skipped a sequence number indicating a lost packet. Furthermore, the manager issues an alarm when a mote resets.

Interference can also be the result of many co-located Dust networks. In the current product line, networks are not synchronized in terms of time or bandwidth allocation so transmissions from one network can occur at the same time and on the same channel as another network. Measures have been taken to reduce the chance that this inter-network interference will cause serious performance issues, but it is possible to see overall path stability be lower in the a multiple co-located networks environment, and is a function of the total combined amount of traffic. Note that lower path stability may not necessarily translate into lower data reliability, severe interference will need to occur for that to happen.

16.2 No Motes Join

Reasons that no motes at all join to the manager include:

- The manager is not running.
- There is no AP mote connected to the manager.
- There is no antenna connected to the AP mote.
- The network ID and/or join key of the manager do not match the security credentials of the motes.
- The Access Control List on the manager does not include any of the motes.
- The motes are all placed too far away from the AP.
- The motes are not powered on.
- The sensor firmware on the motes is not sending the join command correctly.

16.3 A Collection of Motes Doesn't Join

If some motes join and others do not, you have at least established that the manager and AP are functional. Reasons that some motes won't join can include:

- Some motes are placed too far away.
- Max motes on the manager has been reached.
- Some of the motes do not have the correct security credentials to join this manager (network ID, join key, ACL entry).
- The motes that are within range have been configured as leaf nodes.

16.4 One Mote Doesn't Join

If the number of motes that fail to join is small relative to network size (i.e. 1 in 100), then potential reasons include:

- That mote has an RF problem (like it is broken or the antenna is not attached).
- That mote has the wrong security credentials.
- That mote is not powered up.
- Max motes has been reached.
- That mote is placed too far away from the rest of the network.

16.5 One Mote Gets Lost and Rejoins Over and Over

Motes should stay connected to the network indefinitely. Possible reasons for a single mote to join and drop off the network and join again include:

- A power supply problem on the mote is resetting it.
- The RF connectivity to neighbors is marginal.
- The RF connectivity to neighbors is highly transient and unstable.
- The mote is in a location where RF connectivity can be severed and then re-established (like in an enclosure or behind a large obstacle).

16.6 Devices Within Operating Range Have Bad Path Stability

It's probably interference, place them closer together to boost SNR.

16.7 I Need to Install a Repeater but I'm Already at Max Motes

Repeater needed for connectivity: Remove one mote and place the repeater, or rearrange motes to shorten paths.

Repeater needed for 1st hop bandwidth: Cut back on reporting rate, or move a mote from farther out into the 1st hop ring.

16.8 Data Latency is Higher than I Expect

Data latency can be lowered on an individual device at the expense of battery life (for the mote and its ancestors) by shortening the service period in a request but keeping publish rate unchanged. Latency can also be improved network wide by increasing the base bandwidth.

16.9 The Network is Using Paths that Don't Look Optimal

The network continually tries to optimize for lowest power - part of which includes trying new paths periodically. There are other considerations besides path stability that come into play.

17 Application Note: Changing Provisioning Factor to Increase Manager Throughput

17.1 Introduction

Managers guard for short-term changes in path stability through *provisioning*, assigning links to motes as a function of the traffic they generate. By default, the provisioning factor in a SmartMesh network is 3x - for every packet expected to pass through a mote, it gets three links. This allows a device to ride through temporary stability drops down to 33% without risk of its queue filling, which could result in lost data. However, there are a limited number of links available at the manager's access point, so provisioning places a cap on packet throughput.

For applications where manager throughput cap is limiting, breaking the network into smaller subnetworks is the preferred method for increasing total throughput. If this is not an acceptable solution, the customer can modify the provisioning factor to increase throughput, within limits. Dust recommends that provisioning never be set lower than 1.5x - path stability dips to < 70% are not uncommon in customer networks we've observed, and without clear knowledge that the network is operating in a low-noise, low-multipath environment, setting it lower is risky. In general, where paths are > 67%, set the provisioning factor to the reciprocal of the lowest observed path stability.

Note that SmartMesh managers use links for functions other than carrying data traffic, *e.g.* sending Keep Alives with enough retries to avoid a path alarm. Because of this, some motes may have more links to the access point than required by traffic alone. If the access point has reached its link limit, these links prevent any other motes from adding links for bandwidth purposes; it is more difficult to approach the limits discussed below in larger and/or deeper networks.

17.2 Changing Provisioning: IP

IP managers with external RAM have 223 links dedicated to upstream data (150 links when no external RAM is used) on a randomized slotframe from 256-284 slots (270 average). Each slot is 7.25 ms. With 3x provisioning, this is 36.1 packets/s. Setting provisioning to 1.5x achieves a maximum throughput of 72.2 packets/s, but requires that all paths have > 70% stability.

To change the provisioning factor using manager CLI (here 1.5x):

```
> set config bwmult=150

> reset system
```

To change the provisioning factor programmatically, use the *setNetworkConfig<bwMult>* manager API.

17.3 Changing Provisioning: WirelessHART

WirelessHART managers have 737 links dedicated to upstream data on a 1024-slot superframe, where each slot is 10 ms. With 3x provisioning, this is 24.0 packets/s. Setting provisioning to 1.5x achieves a maximum throughput of 48.0 packets/s.

To change the provisioning factor (here 1.5x), you need to modify the *link_oversubscribe* parameter in the `dcc.ini` file in `/opt/dust-manager/conf/config/dcc.ini`.

```
# Oversubscribe coefficient for link (1.0 no oversubscribing)
# Range: 1-100
 link_oversubscribe = 1.5
```

By default, the `dcc.ini` file does not exist. If you haven't already made parameter changes, you'll have to create the `dcc.ini` file to change the provisioning factor. Be sure to create it in the directory listed above.

18 Application Note: Debugging Congested Networks

18.1 Introduction

SmartMesh networks are designed to deliver every packet accepted by each mote from each sensor. If a packet is accepted by a mote but does not make it to the manager, this packet is classified as *lost* and counts against the *reliability* of the network. The reliability statistic that the manager reports is the ratio of non-lost packets to the total number of accepted packets.

There is another important metric called *availability.* Availability is the fraction of times that the sensor was able to hand its packet off to the mote when it wanted to. SmartMesh networks are generally provisioned with enough links to ensure 100% availability in addition to 100% reliability, but we do not directly measure availability because it is an application-layer metric. The only time that a mote is unable to accept a packet is when it has a full queue of packets, we call this a *congested* mote. A congested mote doesn't have enough upstream links to support its local and forwarding traffic. To determine if a network is in danger of losing availability, one must look for congested motes.

18.2 Respecting Services

SmartMesh managers use a *service* model to assign bandwidth in a network. In this model, each sensor application is responsible for figuring out how much data it needs to send and the associated mote is responsible for requesting enough bandwidth from the manager. In short, the responsibility of the application is to:

1. Calculate the packet generation interval for each separate data flow
2. Request a service for the sum total of all these data flows (these can be separate in SmartMesh WirelessHART but only one service in SmartMesh IP)
3. Wait for confirmation that this service, or a faster service, has been accepted by the manager before publishing
4. If no confirmation arrives, re-request the service after a timeout.

There are many details about the service model that are different between SmartMesh WirelessHART and SmartMesh IP. Refer to the respective guides for more information (SmartMesh WirelessHART Services and SmartMesh IP Services).

18.3 Estimating Availability

If you know that a mote is generating periodic data and you know the period, you can estimate how many packets should be received at the manager every 15 minutes. For example, if all motes are reporting once per second, you should expect 900 data packets per 15 minutes. On top of this, the mote sends three health report packets per 15 minutes and may also send responses to manager requests and path alarms. Availability typically is 100% or will drop considerably, so anything around 903 packets per 15 minute interval should be considered good in this example.

When there are fewer packets sent per interval, it gets more difficult to ascertain if they were all accepted by the mote and successfully received by the manager. Networks with less reporting have fewer links so the latency is generally longer which can push packet into the next 15 minute interval and a couple packets plus or minus can be a big fraction of the total number reported. In the following capture of mote statistics, all motes are reporting once per 5 seconds so we expect about 183 packets per interval. Mote 11 has a suspiciously low number in the PkArr column which tells us how many packets arrived during the interval. While the examples shown below show SmartMesh WirelessHART manager statistics, the same concepts apply to SmartMesh IP networks.

```
> show stat short 0
It is now ................. 06/06/12 16:30:17.

This interval started at ... 06/06/12 16:15:00.

 -------------------------------NETWORK STATS---------------------------------

PkArr  PkLost  PkTx(Fail/ Mic/ Seq)  PkRx  Relia.  Latency   Stability

 1806       0  5395(2430/   0/   0)  3098    100%   3.61s      54.96%

 --------------------------------MOTE STATS-------------------------------------

 Id PkArr PkLst PkGen PkTer PkFwd PkDrp PkDup Late. Jn Hop avQ mxQ me ne Chg  T

  2   181     0   181     0   356     0    16  1.34  0   1   0   4  0  0   0 22

  3   182     0   182     0   194     0    20  1.49  0 1.2   0   3  0  0   0 22

  4   182     0   183     1    98     0    17  1.63  0   1   0   4  0  0   0 22

  5   182     0   182     0   329     0    18  1.46  0 1.4   0   4  0  0   0 22

  6   182     0   182     0    82     0    16   1.7  0 1.2   0   2  0  0   0 22

  7   154     0    --    --    --    --    42  20.7  0 2.6   -   -  -  -   -  -

  8   183     0   157     4    43     0    30  2.66  0 2.3   0   1  0  0   0 22

  9   183     0   182     0     0     0    30  3.39  0   2   0   2  0  0   0 22

 10   182     0   188     6    65     0    45  2.96  0 2.1   0   3  0  0   0 22

 11   166     0   169     1     0     0    42  17.2  0   3   1   4  0  0   0 22
```

Reliability here is 100% because there are no entries in the PkLst column, but the application was expecting more packets from motes 7 and 11 so availability < 100% for these motes. In general, you will not know how many packets the sensor is trying to send so you need to look instead for signs of congestion.

18.4 Identifying Congestion

In addition to the estimated availability, congestion can be identified by larger-than-expected latency or by looking at the mote queues in the stats. As a rule of thumb, motes that have an average queue length (avQ) greater than 0 are in danger of seeing congestion at some point. Motes that have a maximum queue length (mxQ) of 4 or more may have experienced acute congestion during the interval. Finally, congested motes typically have higher latency than their peers. Mote 11 in the above stats meets all of these criteria and was definitely congested during the previous 15 minutes.

A missing health report can also indicate congestion. In this scenario, when it was time for the mote to generate a health report, its queue was full and the mote was unable to complete the action. In the mote stats, this manifests like the mote 7 row above. We do not directly get a report on the queue occupancy, but we do still have the lower PkArr than its peers and high latency. When a mote does miss a health report, it continues to keep increasing its counters so that the next health report still summarizes the missing information.

When a mote is congested, it will start sending NACKs to its children when they try to unload their packets. This reduces the effective upstream bandwidth at the children of the congested mote and can, if there isn't enough provisioning margin, lead to the children becoming congested themselves. This process can repeat down through the mesh. Because of this, a mote three hops deeper than the problem mote may get congested and see that its sensor is backing off. This results in fewer data packets from this mote being received at the Manager than expected. In order to find the source of the congestion, the upstream route of each congested mote should be traced towards the AP. The lowest-hop mote that is congested is likely the source of the congestion of its descendants.

18.5 Bandwidth Model

The manager tries to give each mote three times as many links as it theoretically needs to carry local and forwarded traffic in a 100% stability network. This means that the network can operate down to 33% stability, though note that 33% stability often means that it is 100% for a little while and then 0% for a while later which is less conducive to successful multi-hop data collection than a constant two-failure-one-success pattern. If there is congestion anywhere, it means that something has broken down in this model, and one of the following is true for the source of congestion:

- It recently lost a parent due to a path alarm
- Path stability is below 33%
- It ran out of assignable links due to power or memory constraints
- A descendant is reporting more than allowed for its service/base bandwidth level

Path alarms can be monitored by subscribing to the manager notifications and are typically induce very temporary congestion that is resolved if the mote has a sufficient number of good neighbors. If the mote does not have enough good neighbors, the path stability could consistently be low and cause chronic congestion, and this can be due to interference. See the _inc_AN_IdentifyingandMitigatingInterference page for guidance.

A `show mote` command can be issued on the manager CLI to see how close the congested mote is to its link limit. Specifically, you should check these two rows:

```
Neighbours: 27 (max 32). Links: 34 (max 100)
  Links per second: 4.882812 (limit: 11.648407)
```

The indicated number of links here, 34, is safely below the maximum of 100 and the power-induced limit of 11.6 link/s is safely above the current 4.9 link/s. If any mote is approaching the link limit, repeater motes should be added in close proximity to these motes.

Finally, the manager expects that each sensor will respect the service model. If a sensor is going to report at a faster rate than that specified in the base bandwidth for the network, it should request a faster service. Still, there is no guarantee that the sensor will do this properly so it should still be considered as a potential root cause.

To confirm that the mote is not overstepping its requested service level, the `show mote` result can again be used:

```
Bandwidth:
      output         planned   0.3906 current     0.3906
     global service             0.3614   delta    -0.0108
      local service  goal      0.0250 current     0.0250
    guaranteed for services    0.0250 for child   0.3241 Free   0.0415
```

Here we are looking at the local service line. The first value of 0.025 here is the pkt/s that the manager thinks the mote has requested including both the service and the base bandwidth. The current value is the same which indicates that the manager has indeed assigned this bandwidth. The negative value of delta indicates that this mote ostensibly has enough bandwidth to support all local and descendant traffic. At 0.025 pkt/s, the mote should generate at most 36 data packets in each 15 minute interval. The mote stats shown earlier, specifically the PkArr column, can be used to confirm that the mote is not generating more than this maximum level and exceeding its allotment.

18.6 Mitigating Congestion

If your SmartMesh network is congested at several locations, it may be that the path stability throughout the network is too low to function properly. In this scenario, the preferred response is to increase the mote density in the deployment. These additional motes, spread out among the previously deployed motes, can multiply the number of potential paths to choose from and thereby increase the overall path stability. If the new motes are not reporting additional data, this solution does not increase the total egress bandwidth required, and if anything, it should decrease the average power of the existing motes.

If the density of the deployment cannot be increased, we can increase the number of links at the congested motes. If the congestion is global, there are two equivalent ways of increasing the number of links at every mote in the network. Either the provisioning can be increased or the base bandwidth can be decreased. If the congestion is localized to one branch in the network, the motes in this branch can request faster services. When we increase the number of links, the motes with the link increases will have corresponding power increases. Furthermore, any of these increases requires extra receive links at the Access Point. If the Access Point does not have extra bandwidth available for allocation, it may be that the network has to be split into two networks to support the extra links.

19 Application Note: Identifying and Mitigating the Effects of Interference

19.1 Introduction

SmartMesh networks are designed to be able to tolerate interference with minimal performance degradation (compared to a non-interference environment) and only a small increase in power. In a small number of cases however, interference can be strong enough to impact performance, typically manifesting with the following symptoms:

- A large number path stability < 60 %, even at RSSI > -70 dBm
- Upstream latency >> 3x the upstream frame length
- Average queue occupancy >= 1, or max occupancy >= 3
- Reliability < 99.9%

Interference can take the form of an in-band interferer such as an 802.11g wireless router, or an extremely loud out of band interferer such as WiMax. Use of a spectrum analyzer (even an inexpensive WiFi sniffer such as a Wi-Spy) can help identify the region of the spectrum being jammed, e.g.

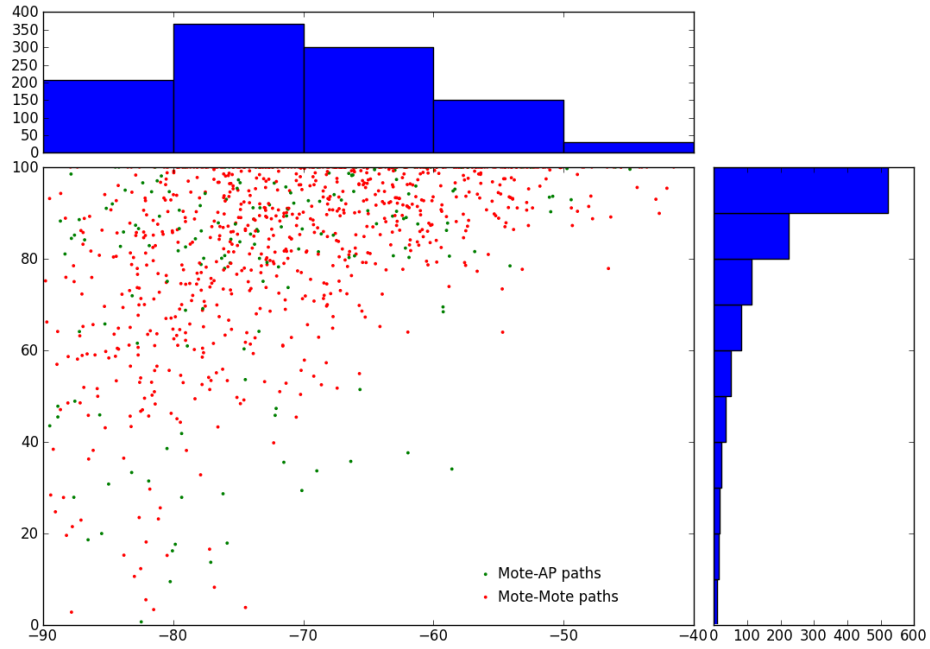
- A fast-hopping interferer such as Bluetooth will appear as a uniformly raised noise floor.
- A 802.11 WiFi router will appear as a broad peak.
- An out of band interferer may not show up at all, but could still saturate receivers in-network.

A directional antenna coupled with a spectrum analyzer may help pinpoint the source of the interference, but it is often possible to deduce the presence of an interferer without using a spectrum analyzer at all - network statistics can often be used to infer the presence of an interferer, and additionally determine if the interferer is significant enough of a problem to warrant mitigation.

19.2 Checking RSSI and Path Stability

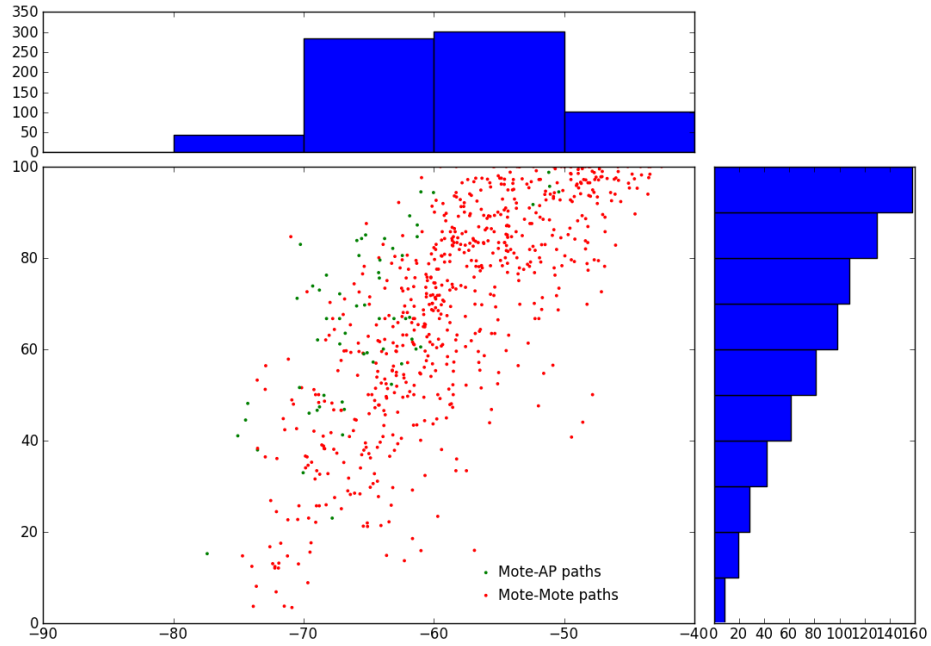
Dust provides a tool that takes a network snapshot file and turns it into a waterfall curve showing the RSSI and path stability.

A good waterfall curve looks like this:



In the top histogram, verify that there are a number of paths used in the -90 to -80 dBm range and also in the -60 to -50 dBm bin. In the right histogram, verify that a significant fraction of the paths are 80% or better and that there aren't many paths below 50%. In the scatter plot, we expect most paths better than -70 dBm to be 60% or higher stability. The waterfall plotter breaks out mote-mote paths and mote-AP paths because sometimes there are hardware or interference differences specific to the AP.

Contrast the good waterfall with this one:



The whole curve here has shifted right and we see very few paths below -70 dBm. This type of waterfall is exemplary of either bad receiver or interference because devices are having a hard time hearing weak signals. There is still a chance that this network could meet customer performance expectations, but special care should be taken to ensure that all motes have sufficient connectivity in a deployment like this.

Each customer should deploy a test network with their specific hardware in a "known good" environment similar to their target deployment environments and without measurable interference. A waterfall curve generated from this test deployment should be used as the best case against which any in-field data can be compared. Antenna choice in particular can have a large impact on receive sensitivity, so it isn't sufficient to use the same best case curve for all customers.

19.3 Checking Mote Latency, Queue Lengths and Reliability

The other checks can be done by looking at the network snapshot file. Here is an example snippet of the short interval stats from the network that had the bad waterfall plot above. Note the line below the "NETWORK STATS" that shows Reliability < 99.9%. The Latency value here is not >> 3x the frame length which is 30 s, but looking underneath the "MOTE STATS" avQ and mxQ columns show that several motes have max queues above 3 and some have nonzero average queues. The presence of lines of missing data is also an indication that the network is in trouble as these motes were unable to generate their health reports due to full queues.

```
< show stat short 2
It is now ................. 04/23/12 15:49:45.
This interval started at ... 04/23/12 15:00:00.
 --------------------------------NETWORK STATS---------------------------------
PkArr  PkLost  PkTx(Fail/ Mic/ Seq)  PkRx  Relia.  Latency   Stability
 1338      13   17k(6603/   0/   0)   13k  99.04%   11.9s      61.44%
 --------------------------------MOTE STATS-----------------------------------
 Id PkArr PkLst PkGen PkTer PkFwd PkDrp PkDup Late. Jn Hop avQ mxQ me ne Chg  T
 10    12     0    10     0     0    35     0  3.25  0 2.6   0   1  0  09154 22
 11    11     0    13     1     0    32     0   1.7  0 1.3   0   1  0  07825 25
 12    17     0    16     2     0    29     2  4.92  0 1.5   0   2  0  08298 24
 19    13     0    16     2     0    39     2  3.76  0 2.5   0   1  0  08429 23
 20    13     0    13     0     0    29     1  6.65  0 1.6   0   1  0  08127 21
 21    12     0    12     0     0    27     4  3.52  0 2.7   0   2  0  08602 21
 22     4     0    --    --    --    --     0 0.171  0   1   -   -  -  -   -  -
 23    11     0    11     2    76     3     2  2.73  0 2.2   0   2  0  0 21k 19
 24    13     0    13     0     0    28     1  1.94  0 1.6   0   1  0  07873 21
 25    10     0    14     5    95     1     2 0.753  0   1   0   4  0  0 43k 18
 26    14     0    13     0     0    28     1   2.6  0 1.4   0   1  0  08060 22
 27    18     0    14     1     0    26     1  5.01  0   3   0   1  0  08085 25
 28    15     0    11     2   111     2     1 0.856  0 1.5   0   2  0  0 28k 19
 29     4     0    --    --    --    --     0 0.792  0   2   -   -  -  -   -  -
 46    22     0    15     0     0    32     2   4.4  0 3.2   0   1  0  07897 28
 47     6     0    --    --    --    --     1  1.16  0   2   -   -  -  -   -  -
 48    14     0    18     6   540     7     4 0.303  0   1   0   5  0  0 78k 18
 56    12     0    11     3    90     0     7  2.81  0 4.3   0   3  0  0 486 19
 57     4     0     5     3   145     0     2  2.75  0 4.5   1   4  0  0 803 19
 64     9     0    13     6   447     7     2  1.54  0   3   2   8  0  01886 19
```

19.4 Mitigation

Once interference has been positively identified, there are a few steps towards correcting the problem:

1. Identify interferer and turn it off or move it. This requires the use of a directional antenna to locate the source, or knowledge of what other devices might be transmitting in vicinity. This is usually only an acceptable solution for the case of a low-power local interferer that is affecting a small section of the network. We've seen one case where a very high power out of band WiMax interferer was identified - it was forced to change operation so as not to interfere with the industrial network it was harming.
2. Blacklisting – if the interferer is know to occupy a small fraction of the spectrum, it can be blacklisted around. Dust systems can be blacklisted down to as few as 5 of the 15 available channels if power is being wasted on fully blocked channels.
3. Increase the provisioning factor - this increases power but ensures reliability and latency don't suffer.
4. Increase the number of parents - this provides immunity against path failures caused by low path stability (the lower the stability, the higher the chance of sufficient random failures in a row to be seen as a failure). Power will increase at all nodes if more downstream parents are added because there are more downstream listens. Adding upstream parents will increase power a little because more Keep Alive packets have to be sent, and will increase latency a little as we are adding more routes in the network in addition to the shortest ones chosen naturally by the manager.
5. Increase the minimum RSSI for a “good” path - this can help speed up joining in interfered networks and can speed up optimization as it will not test out lower-RSSI paths.
6. Add a narrowband filter to reject out-of-band interferers - this is a fix to mote hardware.

20 Application Note: Obtaining Accurate Timestamps

20.1 Time

All devices in a Dust network share a sense of time. This allows a sensor application to use network time to provide for accurate timestamping of sensor measurements. Accuracy can be as tight as 10's of µs under ideal conditions. The process is similar across families:

1. The sensor processor takes a measurement and strobes the mote's time pin
2. The mote returns its local time (in UTC and ASN)
3. The sensor processor places a timestamp and data in a packet
4. Mote forwards packet to manager for delivery to the host application
5. Host post-processes the data notification as needed to account for differences between network time and "absolute" time

Timestamp accuracy of a measurement is determined by various uncertainties in the system, which will be discussed in this application note.

20.2 References

- RFC 5905 defines version 4 of the Network Time Protocol (NTP)
- IEEE1588-2008 defines version 2 of the Precision Time Protocol (PTP)
- IEEE C37.238 is a spec for using 1588 in power systems. It specifies using Ethernet and various settings for synchronizing widely separated installations.

20.3 IP Eterna-Based Systems

A flowchart of the procedure for using network time is shown in Figure 1. The SmartMesh IP manager has a *setTime* API which allows the user to set the UTC time on the manager prior to network formation - the network's ASN counter will then be based on this UTC time. From that point on however, the manager will slowly drift away from "absolute" time, resulting in **Manager UTC uncertainty**, as there is currently no mechanism to continually correct time (e.g. by NTP as is available in the SmartMesh WirelessHART manager). By using the manager's time pin, a host application can correct for this drift. It is not necessary to set a valid UTC time on the manager to make accurate timestamps, since time is returned in both ASN and UTC on the mote.

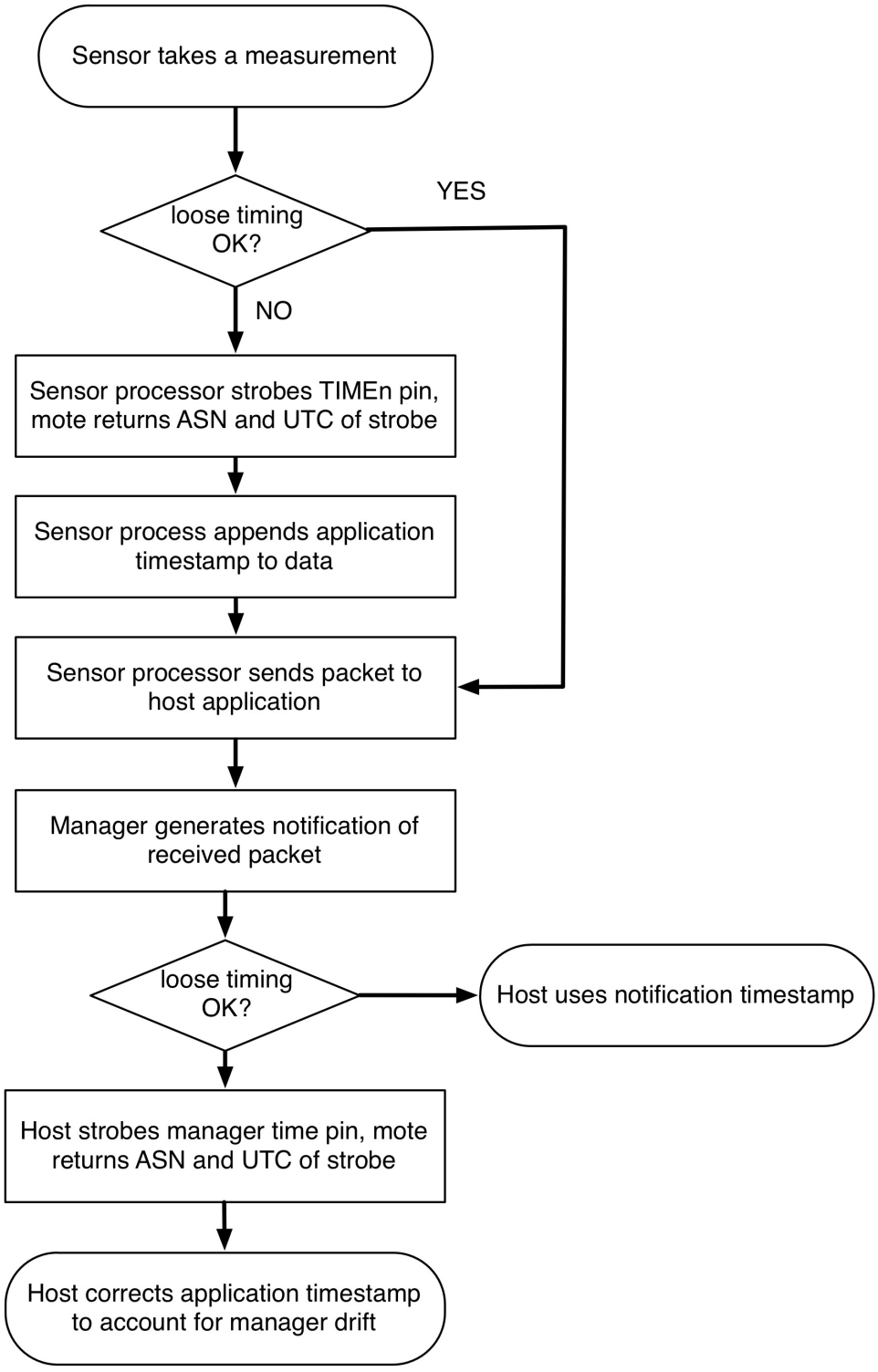


Figure 1 – Timestamping in an Eterna LTC5800/5901/5902 manager network

20.4 Loose Timing

If only loose (100's of ms) timing is required, no application timestamp is required - the host application can use the timestamp from the *data* or *ipData* notification. This timestamp has **queuing uncertainty**, which can be 10's of ms. This timestamp still has Manager UTC uncertainty, which we will discuss correcting in the context of timestamped data.

20.5 Tight Timing

If sub-millisecond timing is required, the sensor processor can take a measurement and strobe the mote's time pin to get a network timestamp to include in the packet. The mote returns its local time (in UTC and ASN), which has two uncertainties - the **mote local time uncertainty**, which is a function of the path stability, temperature variation among motes, topology, etc., and the mote's **time pin uncertainty**, which is a datasheet property for the HW family. Either ASN or UTC timestamp can be included in the packet - using ASN makes it clearer that the UTC time may not in fact be accurate.

When the Application Processor receives a data notification, it must trigger the manager's time pin, thus getting the current ASN or UTC time to within the manager's **time pin uncertainty**. This allows the application to use the timestamp in the packet, along with the current timestamp from the manager to eliminate **Manager UTC uncertainty** and calculate the absolute timestamp of the packet to within the accuracy of time at the host.

20.6 Highest Precision Timing

To get highest precision timing, all sources of uncertainty must be accounted for, and minimized if possible:

- Transit time (latency) adds **transit uncertainty**, which is a function of network topology and activity. If the packet was generated on a very low activity network, it could take 30 s to arrive at the manager – if the manager drift is 50 ppm, manager drift during transit would add 1.5 ms of uncertainty to the calculation. Transit uncertainty can be removed if the application processor periodically strobes the manager's time pin and measures manager drift – then it can calculate a running UTC offset to remove transit time induced uncertainty in addition to Manager UTC uncertainty. Under typical conditions however, transit uncertainty is a few µs.
- Any delay or uncertainty within the sensor processor related to taking a sample and strobing the mote's time pin are up to the integrator to determine and account for.
- A fast temperature ramp can dominate the **mote local time uncertainty**. This error will propagate to all descendants of the ramping mote. In cases where this effect can raise the error beyond the system requirement, the application can measure temperature at all motes and report alarms when significant ramps are detected. The host application can then choose to ignore data timestamped during these alarm periods or assign them more uncertainty than measurements taken during times where all motes sit at constant temperatures. Flat topology networks operating at a stable temperature will have the lowest mote local time uncertainty.

20.7 Quantifying IP Uncertainty

The following are values for the uncertainties in an IP network:

- **Time pin uncertainty** – +/- 1 µs worst case for an Eterna-based manager or mote.
- **Mote local time uncertainty** – for a mote at *h* hops, we expect the typical uncertainty to be $0.6h^{1/2}$ µs and a worst case of 30*h* µs. On top of this, temperature ramping up to our specified limit of 8 °C/min increases each distribution by 1.5 µs·min/°C.
- **Manager UTC uncertainty** – +/- 50 µs/s of up time worst case, +/- 2 µs/s of up time typical if the crystal has been characterized.
- **Transit uncertainty** – +/- 50 µs/s of packet transit time worst case, +/- 2 µs/s of transit time typical if the crystal has been characterized.
- **Queuing uncertainty** – < 50 ms worst case, assuming the packet is acknowledged by the mote. Only relevant if using network layer timestamps.

20.8 WirelessHART (Linux SBC-Based) Systems

The Linux SBC based (PM2511/DN2511/LTC5903) manager supports NTP for keeping the manager synchronized to an absolute time reference. The manager is connected to an NTP server to get UTC (global) time. The accuracy of the manager's notion of global time is characterized by the **Manager UTC uncertainty.** Connecting the NTP server with a dedicated connection could eliminate IP network contributions to uncertainty.

Network time (ASN) is kept separately by the Access Point (AP). The manager measures the AP's time by periodically strobing the AP's time pin, and comparing the resulting time with its own estimate. This measurement contains the AP's **time pin uncertainty**, which is a function of which AP is used (DN2510 or Eterna). When the AP's time differs by > 100 ms (configurable via the *max_utc_drift* ini parameter), the manager pushes a new UTC time mapping to all motes via unreliable broadcast. Reducing *max_utc_drift* results in more messages downstream – for example an LTC5800-based AP with 50 ppm drift would require a DS message approximately every 30 minutes to be within 100 ms of the manager's UTC time. To keep it to 10 ms requires a message every 3 minutes.

If the sensor processor is using UTC timestamps on the mote, then the **UTC mapping uncertainty** must be added to the UTC uncertainty. It is possible for a mote to miss several UTC updates, and thus be off >200 ms from the AP. In theory, using ASN timestamps could eliminate the UTC mapping uncertainty, but the WirelessHART manager does not provide a way to get a precise ASN – the *getTime* manager API returns the last ASN/UTC mapping that the manager measured, and so could be up to *max_utc_drift* ms in error.

Since many of the sources of uncertainty cannot be controlled or accounted for in a current SmartMesh WirelessHART manager, only loose timing (100's of ms) is available.

20.9 Quantifying WirelessHART Uncertainty

The following are values for the uncertainties in a WirelessHART network:

- **Manager UTC uncertainty** – this is a function of the manager, the NTP server, and IP network latency. It is not a function the AP. This has been measured to be ~1 ms on low-traffic Ethernet networks, but can rise to many 10's of ms when the Ethernet network becomes busy.
- **Time pin uncertainty** – this is expected to be -60/+600 µs for a DN2510 based AP or mote, and +/-1 µs for an Eterna-based mote.
- **UTC mapping uncertainty** - It is possible for a mote to miss several UTC updates, and thus be off up to +/- 300 ms from the AP.
- **Mote local time uncertainty** – With a DN2510-based AP, we expect the mean to be around $45h^{1/2}$ µs and a worst case of $150h$ µs for a mote at h hops.. With an LTC5800-based AP, we expect the mean to be $1.2h^{1/2}$ µs and a worst case of $30h$ µs. On top of this, temperature ramping up to our spec limit of 8 °C/min increases each distribution by 3 µs·min/°C.

20.10 Synchronous Events

Synchronous measurements or events are also possible with all products, with a slightly different procedure:

- The application uses the manager *getTime* API call to get current ASN.
- The host application broadcasts* an application-layer packet containing an event time (in the future). The message is sent 3 times to ensure delivery**. The future ASN can be used by all the motes to synchronize an event. A large enough future ASN should be chosen to ensure that every mote receives the future ASN and has time to prepare for the event - any value greater than 120 seconds is safe.
- Motes receive the packet and send a notification to their sensor processor containing the application-layer packet.
- Each sensor processor uses the mote time pin to determine the current time.
- Each sensor processor determines when it is time for the event and takes a measurement, or actuates, as appropriate. Assuming that the sensor processor is running an uncompensated 32 kHz xtal as a time reference (+/- 150 PPM), device to device variation in the absolute sample time on a 120 second timer could be off by +/-15 ms. Note that this requires a 24 bit timer. If this level of synchronization is unacceptable, the sensor processor can periodically poll the mote for the current ASN and adjust its timer accordingly. Conversion between ASN delta and time is:

delta time = delta ASN * slot length

where slot length for SmartMesh WirelessHART is 10 ms, and for SmartMesh IP is 7.25 ms.

- Each sensor processor can optionally execute the timestamp logic above for any data generated by the event.

*The destination (IP mesh layer or WirelessHART net layer) address is 0xFFFF**3 retries is chosen for unreliable downstream transport as a tradeoff between wait time and likelihood of all the motes receiving the command. More retries increases this likelihood at the cost of a longer wait (i.e. the future ASN must be farther out). Selective retries (to reduce duplicates) are possible if application level acknowledgements are provided by the sensor processor. The sensor process must discard duplicate commands (containing the same future ASN).

21 Application Note: Using Multiple Managers to Build Large Networks

21.1 Large Deployments

There are installations where there will be more sense points than can be supported by a single manager - we handle this today by placing multiple managers to cover the installation. Some or all of these managers and their motes are expected to have overlapping radio coverage. When possible, it is good practice to separate "co-located" managers by a few meters instead of placing them right next to each other - this is to avoid radio interference between their Access Points. Installations with multiple co-located managers will function perfectly well provided the total traffic in the vicinity is kept at a safe level.

Each manager controls its own network, and there are two critical ways that motes are separated into various networks. First, each network has a Network ID. This value is in every advertisement sent by the network and is used to filter packets once motes have joined. Second, the manager can have an Access Control List (ACL) which specifies the MAC address and join key of each mote that is allowed to join the network. By using Network ID and ACL together in different ways, motes in a large deployment can be partitioned into smaller networks. Controlling Network ID puts the onus on the mote to decide which network it is going to join. Controlling the ACL allows the manager to make the decision, but this can be bad for a mote trying to join a network for which it is not on the ACL.

Depending on the software license used, the SmartMesh WirelessHART manager can support up to 250 or 500 motes and the SmartMesh IP manager can support up to 32 or 100 motes. Obviously, a deployment requiring more motes than this requires multiple managers, but multiple managers can also be deployed to increase the overall egress bandwidth of the deployment. Each SmartMesh manager supports 20-25 packet/s of egress bandwidth (as high as 40 packet/s if provisioning is reduced), so for example, a deployment with 100 motes reporting data once per second could be safely done with five SmartMesh managers.

21.2 RF Limitations

For SmartMesh WirelessHART, there are 15 channels and 100 slots per second. This is enough *cell space* for an absolute maximum of 1500 packet/s to be transmitted on different channels and/or at different times. When multiple managers are sharing the same radio space, there will be collisions if traffic overlaps because the managers do not share their schedules or their precise sense of time with each other. Empirically, we find that it is safe to use about 25% of the cell space when co-locating managers, so in this case it is safe to add managers up to the point of 375 packet/s total egress bandwidth. As an example, suppose 1800 motes are deployed using four SmartMesh WirelessHART managers and each mote is configured to report temperature every 30 seconds. The total egress requirement here is 1800/30 = 60 packet/s so this deployment is well under the RF limit.

In SmartMesh IP, there are the same 15 channels but there are 138 slots per second. Running through the same calculations, the safe limit for co-located managers is 517 packet/s.

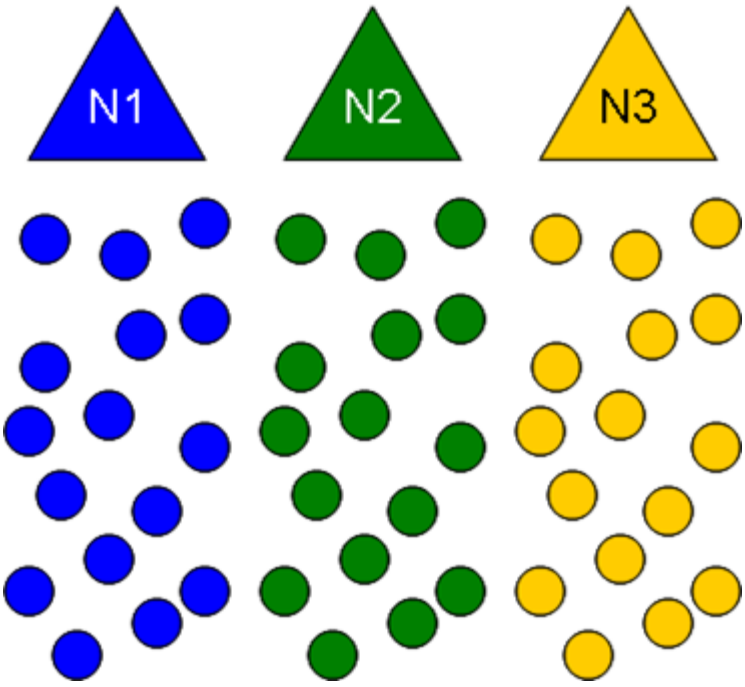
As traffic is added to overlapping networks, inter-network collisions will manifest as small decreases in path stability. SmartMesh managers have mitigation strategies to spread out the collisions so that they tend to affect all paths equally instead of completely killing any one path. Because Dust networks are low-duty cycle and low-power, interference from other Dust networks is less damaging than other forms of interference like WiFi.

21.3 Ideal Deployment Guidelines

In the ideal case, motes are divided equally among the available managers. If there are 48 motes and 3 SmartMesh managers, for example, then each manager is given 16 motes. To accomplish this, there are several steps:

- Three unique Network IDs are chosen and one given to each manager
- Sixteen motes are designated for each manager and programmed with the appropriate Network ID
- Each manager is programmed with an ACL containing its 16 mote MAC addresses and join keys
- In the field, the motes are placed so that each 16-mote network follows our deployment guidelines

In the figure below, these three networks partitioned by Network ID are represented by different colors. The motes are physically placed in locations such that the each colored network is well connected independent of any other network's motes.



Motes filter advertisements by Network ID, so a joining mote will only try to send a join request to a parent with the same Network ID as the joining mote. If a mote has the correct Network ID but the manager does not have that mote in its ACL (a mistake in this scenario), the mote will try to join but the join request packet will be dropped by the manager. In this case, the mote takes several minutes before exhausting its retries and resetting.

While this approach is the most secure and speeds up network formation, the optimal performance comes at the cost of configuration time. Each mote must be programmed to the correct Network ID, and installed in the right area on the site. Each manager must be populated with the correct ACL. If a replacement device or a repeater is needed in one of the networks, the replacement mote must be programmed to the correct Network ID and the correct manager must be given an ACL entry before the new device can join. Furthermore, if a mote moves from one location to another in the deployment space, it may leave its current Network ID and not be able to rejoin using a different Network ID for its new location.

21.4 Large IP Networks - Different Network ID and Common Join Key

As best-practice security, we recommend always using an ACL and unique per-mote join keys. However, current SmartMesh IP managers are extremely memory-limited and can only store ACL entries for up to 32 or 100 motes (if external RAM is used). So for single-manager deployments we recommend using best practices, but for multi-manager deployments this is not possible unless it is known a-priori which manager each mote can hear (and will continue to be able to hear.) We recommend not using an ACL, using a common join key, and separating out motes into networks using different Network IDs. We still recommend using a deployment-specific join key shared by all motes but unique to the customer so as to still disallow malicious devices from joining the network. This means that any mote in the deployment can join any network providing it knows the secure common join key; it is up to the mote to select the appropriate Network ID for the network it wants to join.

The mote application can use the *search* API prior to joining to scan for different Network IDs in the vicinity. In this mode, the mote listens to **all** advertisements regardless of the Network ID. For each advertisement it hears, the mote reports the Network ID of the sender, the signal strength, and the depth of the sender in hops. This gives the application enough data to intelligently select which network it wants to join. The application then sets the Network ID of the mote and issues the *join* command which tells the mote to start listening to advertisements for its particular Network ID. The mote then tries to join only this prescribed Network ID. We recommend programming the application with a preferred Network ID that the mote should try when first booting up. If no advertisements from this Network ID are heard after a timeout, the application can set the mote to *search* mode to gather the information to join an alternative network. This process also allows some mobility; motes that move and reset with neighbors having a different Network ID can discover this new Network ID and rejoin a different network in the multi-manager deployment.

Future versions of SmartMesh IP managers will have more ACL entries to provide additional security in large deployments.

21.5 Large WirelessHART Networks - Different Network ID and Shared ACL

SmartMesh WirelessHART managers can store thousands of mote MAC IDs and join keys in their ACL. If a multi-manager deployment is being planned, say with 1000 motes and 4 managers, each manager can be given the full 1000 motes worth of ACL and join keys, and its own separate Network ID. The mote applications can then use the *search* API to scan advertisements from all networks in the area as described in the previous section.

Alternatively, a multiple-manager deployment can be done with the same Network ID and with different ACLs. In this case, each manager must know which motes it is allowed to accept. The only justification for using this method is in the case where it is desirable to have motes join predefined managers. For example, if 300 motes are being split evenly among 6 SmartMesh Wireless HART managers, each manager could be given a different 50-mote ACL that would determine which motes end up being controlled by which manager. The downside of using this strategy alone is that a mote trying to join the wrong manager will not know that it is being denied. This mote will send in a join request packet through the parent that it heard advertising, and it will get a proper link-layer ACK from that parent. The joining mote will maintain synch with the network while waiting for a response from the manager, but the manager will drop the join request because the joining mote is not on the ACL. It takes a long timeout period before the mote will reset and try joining again. There are cases, however, where it is not easy to set Network ID for motes once they have left the factory and ACL separation is the only way to achieve balanced networks because all the networks must run using the same Network ID.

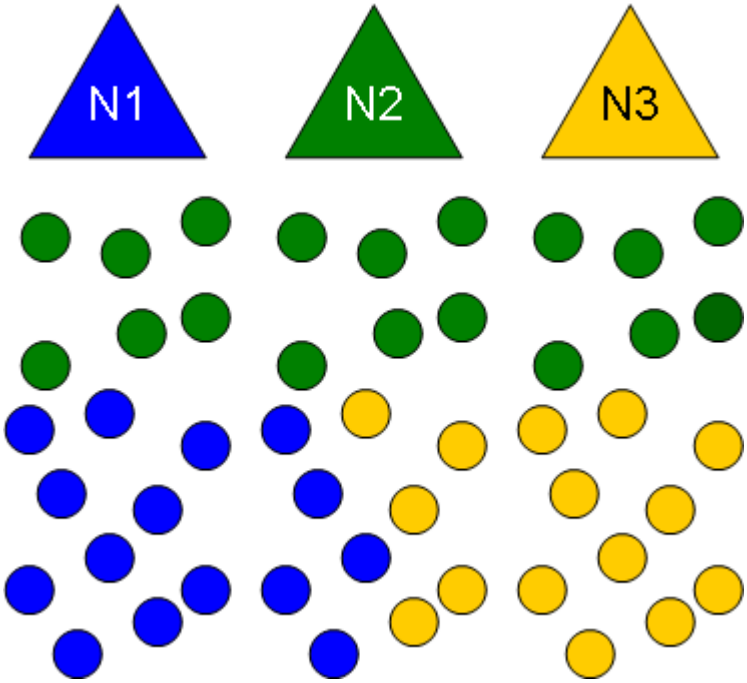
Providing that each network follows the deployment guidelines for range and connectivity, using ACL separation alone will eventually result in all motes ending up in their prescribed network, it may just take awhile. If a mote moves to a new area of the deployment outside of its original network, it cannot rejoin unless the manager of the new area is programmed with a new ACL entry. To enable mobility for a small number of motes, all managers in a deployment can have an ACL entry for these mobile motes. A mobile device leaving a network will then reset and rejoin a new network after hearing new advertisements.

21.6 Multiple-Manager Deployment Risks

There are a couple risks that should be evaluated when deciding which approach to take.

It is also possible to set all motes and managers to a single Network ID and not separate out the ACLs at all. If a single Network ID is used, motes will greedily join whichever network they happen to hear. This can result in motes not being assigned equally to the available managers as the manager that starts advertising first tends to start an avalanche effect of joining. This popular manager could reach its mote or bandwidth maximum having joined all the motes close to a clump of multiple managers and this could strand more distant motes. If this approach is taken, the application must be able to recognize that managers are filling up and reset motes intelligently in hopes that they will join under-used managers.

Before leaving the factory for deployment, the individual networks could be partitioned properly using separate Network IDs and ACLs and put into separate boxes for network 1, network 2, etc. But care must be taken during deployment to locate the motes properly as well. The individual deploying the motes might put all the motes from the network 1 box near the managers, stranding motes for network 2 beyond the range of the manager for that network. Even if some attempt is made at spreading the network out, care must be taken so that the connectivity deployment guidelines are followed for each network individually, not just for the deployment as a whole. Special attention should be paid to bottlenecks - there may be many motes around a mote from network 1 but they may all be from other networks. In this case, a follow-up visit is sometimes required to install repeaters, and these repeaters must be properly programmed with the specific Network ID.



Improper separation of the networks can lead to bottlenecks or starvation as shown in the above figure. Here the green manager has taken all the motes close to the manager and the yellow and blue managers are not close enough to their motes to allow them to join.

21.7 Setting Network ID and Join Key Over the Air

From the manager, the application can set the Network ID and join key on the motes over the air. Some customers use a fixed Network ID and join key during a test build at the factory and then reprogram the settings to unique values prior to packaging up the network for deployment. This is particularly useful for customers with sealed mote packages that cannot be physically connected for reprogramming. The new values take effect after a mote/manager reset.

22 Application Note: Using the SmartMesh Power and Performance Estimator

22.1 Introduction

This document describes some of the ways you can use the SmartMesh Power and Performance Estimator to make high level conclusions about how SmartMesh Networks operate. Many of these conclusions are not necessarily intuitive, but once you understand them, you will be much better equipped to make system-level decisions on how to plan out wireless sensor networks based on SmartMesh products. The analyses below use the SmartMesh IP platform, but the techniques hold equally true for SmartMesh WirelessHART networks, which are covered with the same spreadsheet.

22.2 Problem: What is my Battery Life?

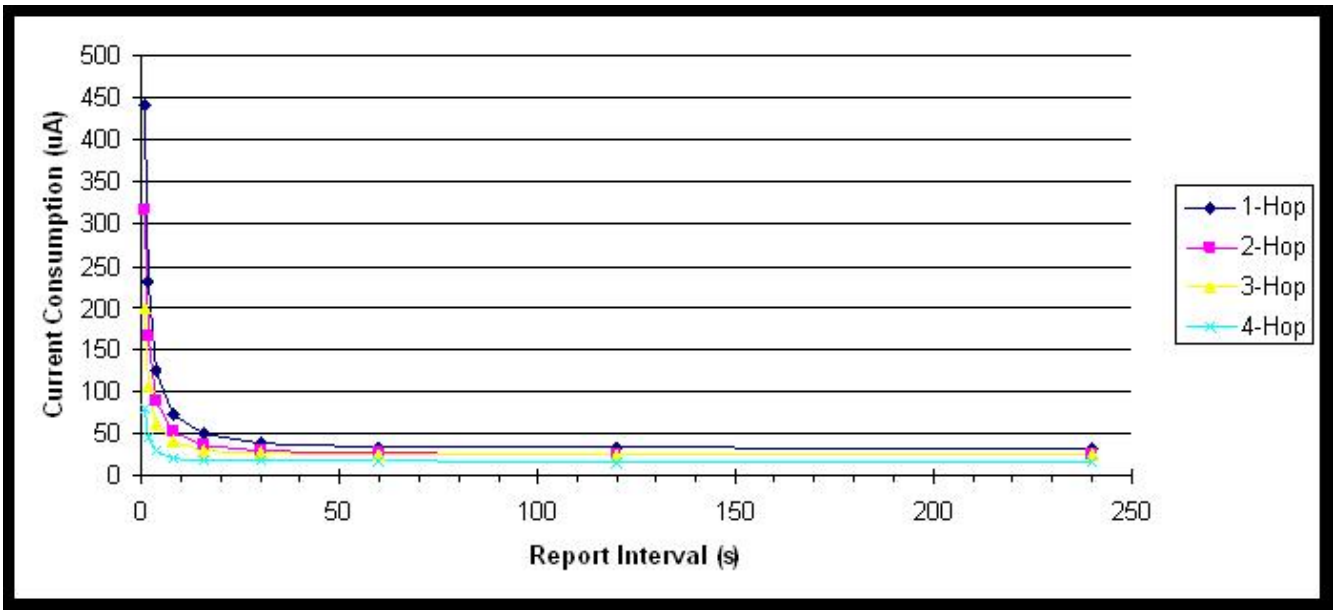
In the wireless sensor network WSN marketplace, battery life or power consumption is a key differentiating feature. SmartMesh products are uniquely well positioned to win in any battery life comparisons because they use the lowest-power radios and the best protocols to deeply duty cycle those radios. That said, it is a fact that at the moment one installs a device, they do not know what the battery life of that device is going to be. Will it be a leaf node in the mesh, responsible for no routing of any other device's data? Will it be a heavily loaded router? Will the retry rate in the network make it work harder? What is the cumulative effect of these uncertainties? Does it make battery life drop in half? By 5x? By 100x?

22.3 The Power and Performance Estimator

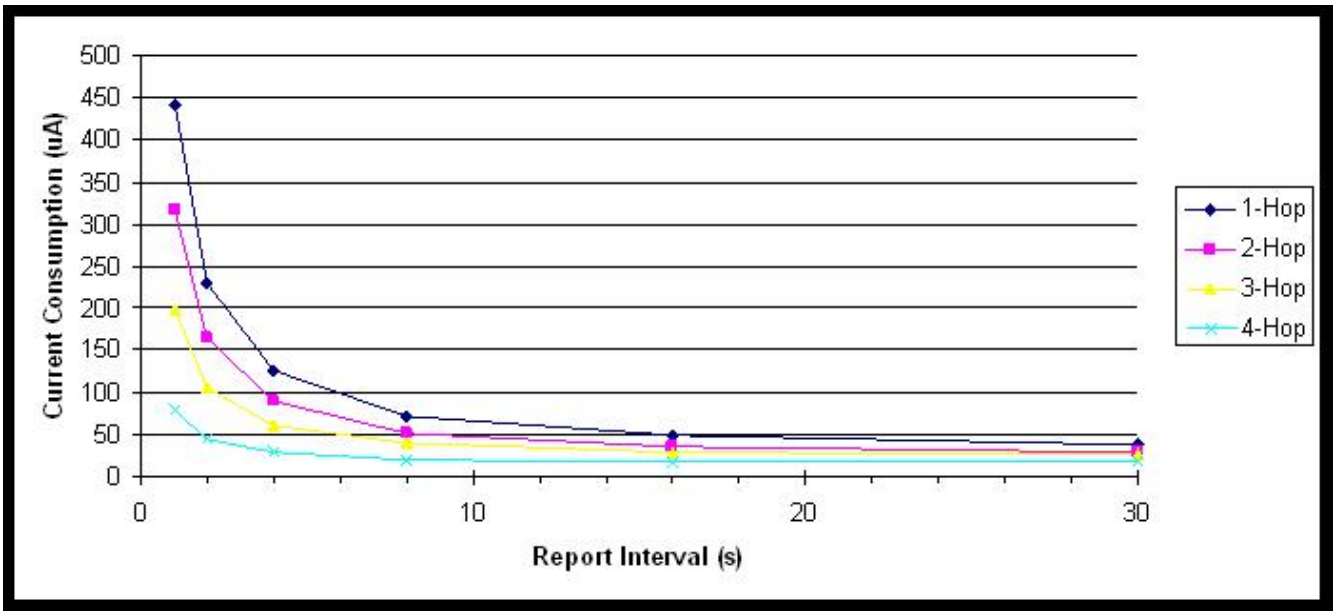
In order to take away some of the uncertainty associated with power consumption and battery life, we've developed the SmartMesh Power and Performance Estimator. You can interact with this Excel spreadsheet to get estimates of power consumption and battery life of motes running in networks that you can specify. The SmartMesh Power and Performance Estimator is an approximate tool for estimating current consumption and battery life, but it is an excellent tool for doing a sensitivity study and for making relative comparisons. The conclusions drawn here are based upon these relative comparisons. In all of the examples, we're using the default values of a 3.6V supply and a room temperature of 25 degrees Celsius.

22.4 Q1: How Does Reporting Rate Affect Power?

Spreadsheet Exercise: Define a network and back off the report rate until you find the minimum possible power consumption for all motes. I'm going to do an IP network with 20 motes, 5 motes at each of 4 hops. I'm going to set the data payload size at 80 bytes, and set the report rate at one packet per second. As a result, the one hop motes are consuming 441 µA. Using a battery like the Tadiran TL4903, which has a nominal capacity of 2400 mA-hr, that translates to a 7 month battery life. The four hop motes, which have no children in the mesh, have about a 3.1 year battery life. Now I repeat at 2 second reporting, 5 second reporting and beyond. The result is as follows:



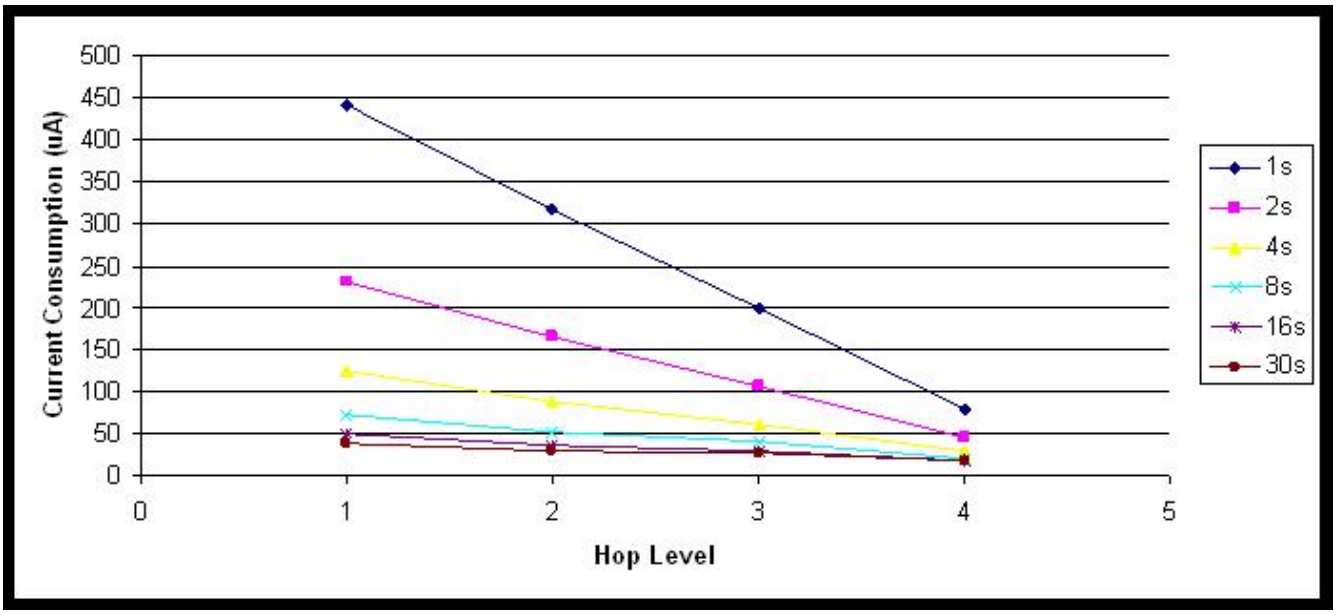
If I back off data reporting from one packet per second to one packet per 2 seconds, I nearly cut current consumption in half, practically doubling battery life. At slow report rates, it doesn't matter much. Whether my sensors report data at once per minute or once per 4 minutes, the current consumption stays flat. If we zoom in on the 1 s through 30 s range we see the following:



It is very easy to build a multi-hop mesh where all devices consume less than 100 µA. At 100 µA, that corresponds to 2.5 year battery life on the TL4903 battery. The conclusion that is important is that if fast reporting is required, ultra-long battery life is difficult. If absolute maximum battery life is required, there is little benefit to reporting data slower than once per minute.
Conclusion 1: There is a minimum report rate below which it doesn't matter. In a SmartMesh network, there is some quantity of radio traffic needed to maintain time synchronization across the network. If sensors NEVER send any data, the traffic in the network will be dominated by this time synchronization ("keep alive") traffic. This sets a floor on the minimum possible power consumption in a network. Any time a sensor sends data, that represents a time where this time synchronization traffic need not be sent.

22.5 Q2: How Much Does Routing Cost?

Spreadsheet Exercise: Using the same data set as above, we can show the cost of routing. In this four hop mesh, the four-hop motes only send their own data. The three hop motes send their own plus the data from their children in the mesh. The one-hop motes are responsible for forwarding all the traffic in the network, plus their own. Slicing the data at various report rates we can illustrate the cost of routing.

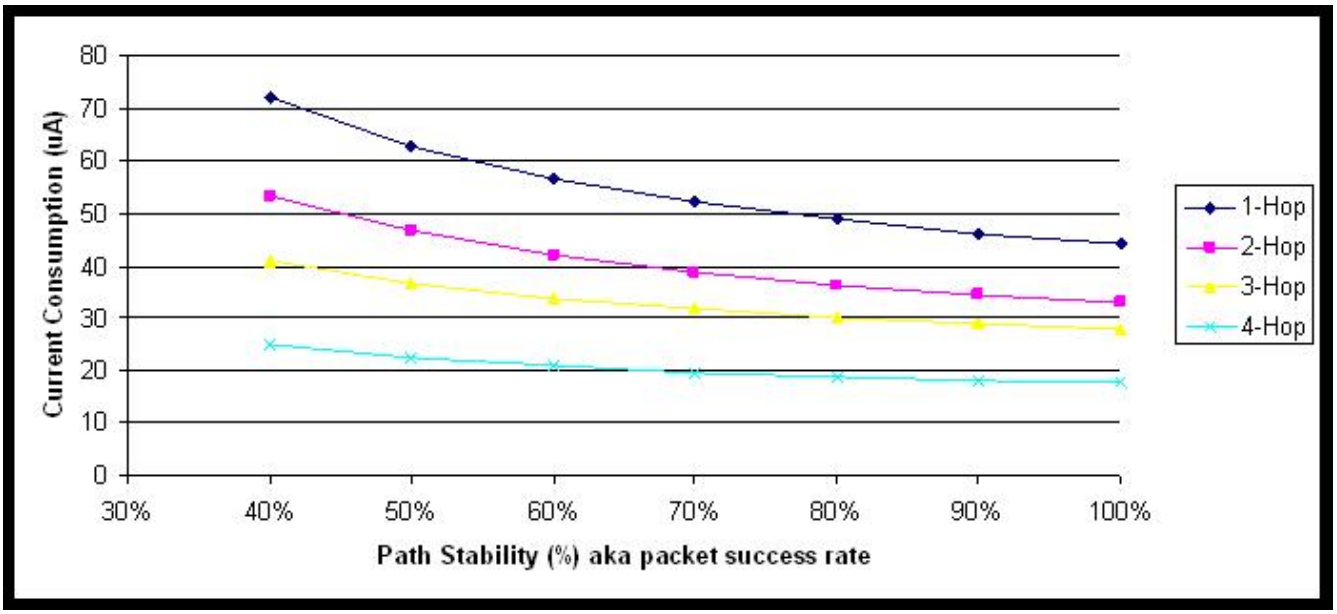


At lower report rates, the cost of routing data gets flatter. The 4-hop motes are always the lowest power, but the devices doing all the routing are not always at several times the current consumption. At moderate report rates, it is possible to build a network with very flat current consumption and reasonably uniform battery life.

Conclusion 2: It costs something to be a router, but not as much as you might expect, particularly as reporting rates approach the keep alive interval.

22.6 Q3: How Much Does Retransmission Cost?

Spreadsheet Exercise: One of the inputs in the spreadsheet is Path Stability. This means the packet success rate or "one minus the retry rate". Some think that optimizing the radio protocol to reduce bit errors and packet errors is the key to reducing power consumption. Retry rates of ~30% are not uncommon in these networks, but does that mean that batteries are dying 30% faster than they should? The way we illustrate that is by keeping our 20-device 4-hop network reporting at 16 s. I vary path stability. The results are as follows:

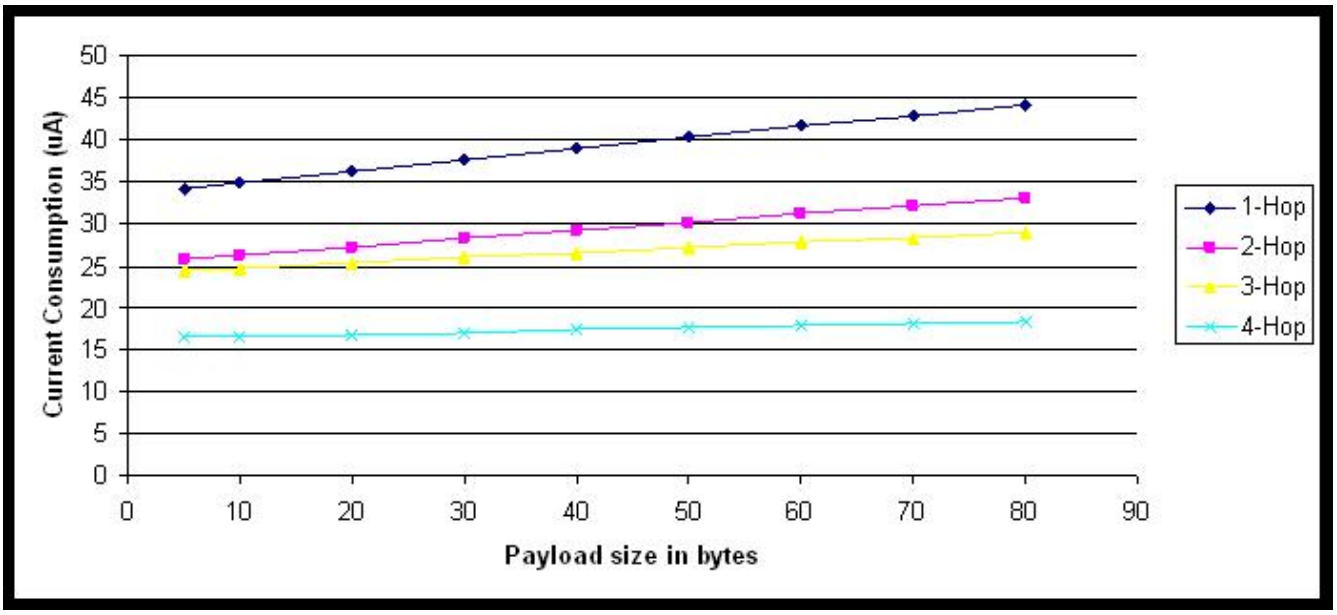


At this moderate report rate, it is a fact that high packet success rate reduces power consumption. At 40% packet success rate, you might think that you will have 2.5x the current consumption and less than half the battery life, but that is not the case. It is certainly better to have strong RF paths with fewer retries needed, but doing a retry should not be considered fatal to battery life. SmartMesh managers understand the tradeoffs between path stability and hop number and optimize accordingly. Sometimes it is better to choose a parent that is closer to the AP than one with higher path stability. This may cause more overall retries in the network, but each packet then has to travel fewer hops to its destination.
We do not expect networks to function well below 40% packet success rate. At very low packet success rates, we find that paths are at the edge of failing completely. The problem there is losing devices entirely from the mesh.

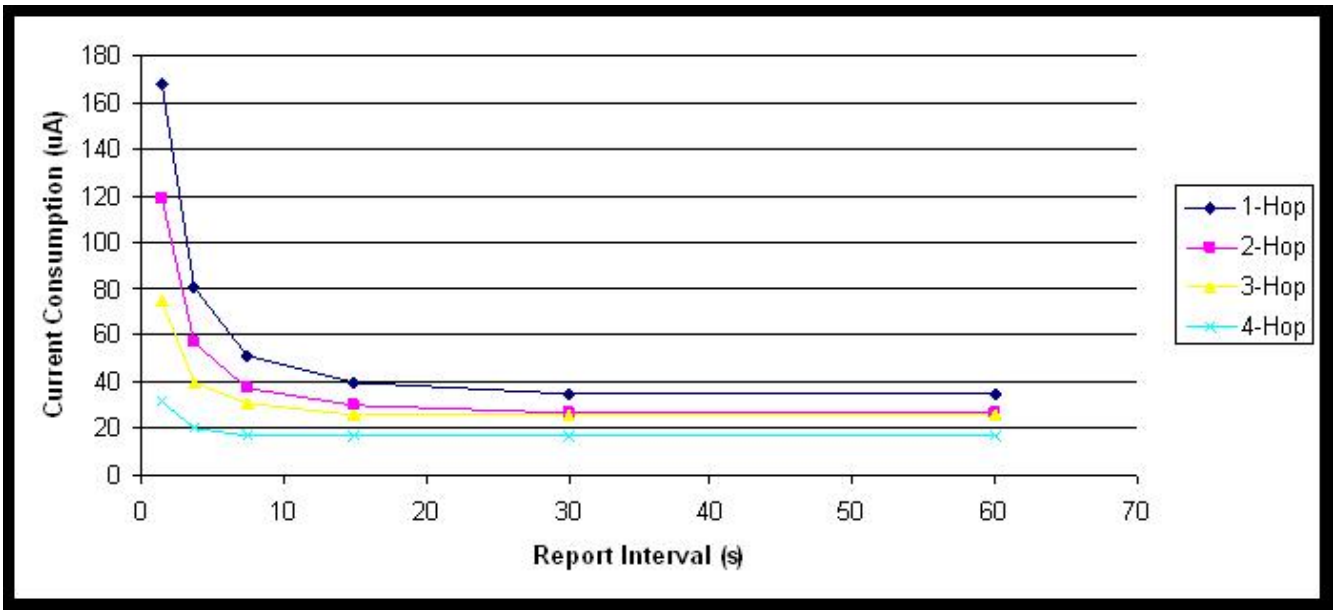
Conclusion 3: Retry rates matter, but not as much as you might think .

22.7 Q4: Can Packet Aggregation Save Power?

Spreadsheet Exercise: In the same 20 mote, 4 hop deep network, we choose one report interval, 20 seconds, and vary the payload size from 5 bytes to 80 bytes. The results are as follows:



It is true that sending a larger data payload increases power consumption, but sending 18x the data in a 90-byte payload only costs about 10% more in current consumption. Looking at it another way, we consider sensors that all send 80 bytes per minute. One sensor application sends an 80 byte payload once per minute. Another sensor application sends two 40 byte payloads, one each 30 s. The third sensor application sends 20 byte payloads every 15 s, and the fourth sends 10 byte payloads every 7.5 s. The fifth sensor application sends 5 byte payloads every 3.75 s. The sixth sensor application sends a 2 byte payload every 1.5 s. The results are as follows:

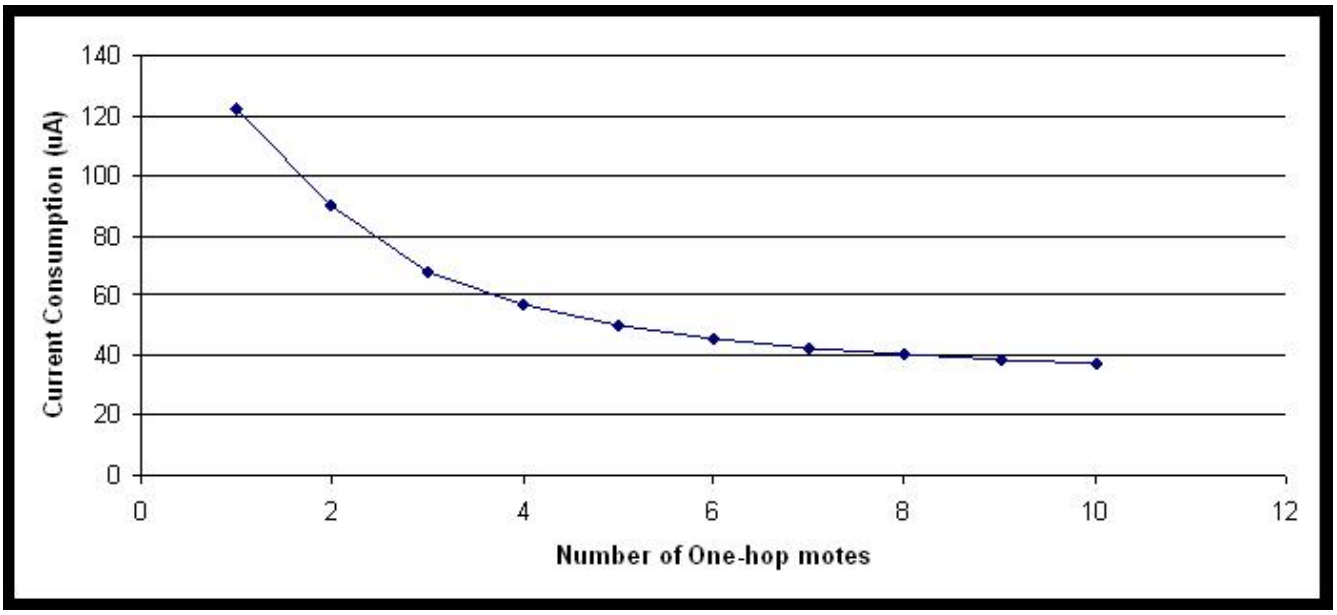


All the above are sending the exact same sensor throughput. Sending larger payloads less often is simply a far more efficient way to transfer data. The trade-off of course is latency. Some applications have no use for old data when a newer reading is available. Other applications can use historical data to make very good decisions. There is a ~4x power consumption and battery life benefit that can be obtained if an application can efficiently use the payload capacity available.

Conclusion 4: It is more efficient to send fewer large payloads than it is to send more small payloads. Also, sending a big payload does not cost much more than sending a small payload, since there is a fixed overhead for all packets.

22.8 Q5: How Does Network Depth Impact Power?

Spreadsheet Exercise: We revisit this same 4-hop mesh, but now we vary the number of one-hop motes. Think of it as a 12-mote cluster of sensors at hops 2 through 4, and the 1-hop motes are just repeaters that the customer is reluctantly buying to bridge the gap from the sensors to the gateway. I have the sensors all sending an 80-byte payload once every 15 seconds. We start at 1 one-hop mote, and keep adding them until we get to 10 one-hop motes. We plot the worst case power consumption as a function of one-hop motes. The results are as follows:

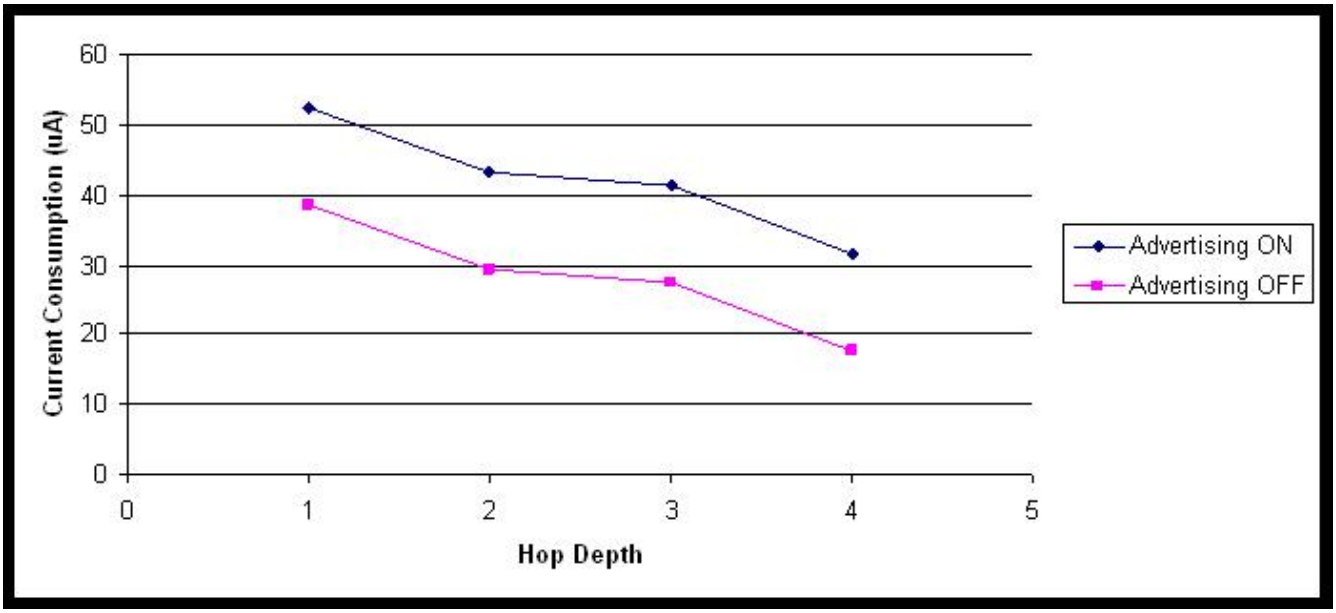


For many systems, the concept of 'battery life' means the moment that the *first* sensor disappears due to a battery dying. At that time, all batteries are replaced. Since the one-hop ring has to forward all the traffic, the first battery to die will almost certainly be in the one hop ring. The easiest way to extend that period of time is to have many one-hop devices. In addition to this effect, having many one-hop devices offers additional robustness vs lost devices. If a network has only two one-hop devices and one is removed, there can no longer be a good mesh at that final hop. Reliability may suffer.
Generally speaking, traffic is increasing at every hop level towards the manager. At each hop level, the motes at that hop level are forwarding all the traffic from descendants, plus adding their own. At each hop level closer to the gateway, the best that can possibly happen in terms of current consumption is for all the devices at that hop level to share the work equally. The more devices there are at that hop level, the lower this equal share can be, and the smaller the penalty if one of these devices goes away.

Conclusion 5: Use as many one-hop motes as possible. Current in any hop ring is inversely proportional to the number of motes in that hop ring.

22.9 Q6: Turn Off Advertising in an IP Network to Save Power?

Spreadsheet Exercise: Note there is an adder for turning ON advertising. In the IP networks, advertising is ON by default. The application must turn it OFF if power savings is desirable. But BE CAREFUL. A new mote cannot possibly join while advertising is off. So, if the application wants to turn it off, it should have mechanisms to turn it back on again. For example, if you form a network, turn off advertising, and then power cycle one mote, it will NEVER join again, unless you turn advertising back on. If the application automatically turns on advertising every time a mote gets lost, then there will be no problem. Also, if the application has a user interface to turn on advertising, then the user can add new devices easily as well. If the application developer is careful, then advertising can be off virtually all the time. This represents a savings of 13.8 µA on every single mote. Looking back at our 4-hop, 20-mote network with 30 s updates, we get the following power curves:



By dropping from 31 µA to 17.2 µA, the leaf nodes could have a battery life of 14.5 years on Tadiran AA cell (2160 mA-hr).
Conclusion 6: Turning off fast advertising when you don't need it is valuable, but be careful.

23 Application Note: What to Expect with Motes That Move

23.1 Moving Motes

In current SmartMesh products, a mote traveling beyond the range of its parents needs to reset and rejoin the network with new parents. Similarly, a new mote arriving in the wireless range of an existing network needs to join the network before it can send and receive data. The exact behavior in these conditions depends on which product is being used.

Motes that move continuously around the network and move further than one radio range are not recommended.

23.2 SmartMesh WirelessHART

For SmartMesh WirelessHART, once the network has transitioned out of Building phase to Steady-State phase, advertising is reduced at the motes to conserve energy: the advertising timer is increased from 1.28 s to 20 s on the motes. This means that a mote appearing in range of only mote advertisers during Steady-State will take about 16 times longer to synch up than during the Building phase. AP advertising does not change from the 0.16 s interval, so new motes that appear within range of the AP will not have any different joining behavior at either of the two states. The application can also use the API to speed up advertising again if it is expecting a new mote to arrive.

A mote already in the network that moves has to lose both of its parent paths before resetting. The path alarm timer is 4 minutes in the WirelessHART standard, and getting these path alarms is the mote's trigger to delete a parent path. As such, we expect a moved mote to take a little over 4 minutes to reset. Once the mote resets, the manager will try to reach the mote through its previous downstream links. Once this state machine times out, which could be up to 15 minutes, advertising will be automatically sped up at the remaining motes to search for the lost mote. In the meantime, if the mote hears advertisements at its new locations, it can rejoin the network before the manager detects that it is lost. During periods of fast advertising, either during building or searching for a new mote, Eterna motes use an extra 16 µA and DN2510-based motes use an extra 30 µA.

For more details on the expected time to join, consult the Network Formation section of the SmartMesh WirelessHART User's Guide.

23.3 SmartMesh IP

For SmartMesh IP, advertising keeps the same period unless explicitly deactivated by the application through the API. Each SmartMesh IP frame is about 2 seconds long; motes advertise once per frame and the AP advertises four times per frame. If advertising has been deactivated, no motes advertise at all, meaning that new motes cannot join and motes cannot rejoin the network. We recommend deactivating advertising only when energy conservation is critical and the application is savvy enough to control it without manager intervention. Deactivating advertising saves 14 µA for Eterna motes.

A mote already in the network that moves has to lose both of its parent paths before resetting. The path alarm timer is 1 minute in SmartMesh IP, and getting these path alarms is the mote's trigger to delete a parent path. As such, we expect a moved mote to take a little over 1 minute to reset. Once the manager fails to reach the mote with downstream queries, it will be marked as **Lost** but no automatic changes to advertising occur. Meanwhile, a moved mote is free to rejoin whenever it hears new advertisements.

For more details on the expected time to join, consult the Network Formation section of the SmartMesh IP User's Guide.

23.4 Summary of Differences Between SmartMesh WirelessHART and IP

For this simple comparison, we assume a path stability of 80% and a join duty cycle of 5%.

Parameter	WH	IP
Time to reset	4 minutes	1 minute
Time to speed up advertising	15 minutes	N/A
Mean Steady-State synch time (1-hop)	60 seconds	188 seconds*
Mean Steady-State synch time (multi-hop, 3 neighbors)	42 minutes	250 seconds*

*If advertising is explicitly turned off by the application in SmartMesh IP, no motes are able to synch up and join the network.

24 Application Note: Migrating Motes Between Networks

24.1 Migrating Motes

Many users will gradually add more motes to a network over its lifetime. At a certain network size, the manager may reach its mote or bandwidth limit and it may be desirable or necessary to move some motes to a new network. This document describes how the customer application can use mote features to intelligently manage this process. We discuss this in the context of a SmartMesh IP network, but the procedure is equally applicable to a SmartMesh WirelessHART network.

24.2 Procedure

A network is deployed in an oil drilling field. As new wells are drilled, motes are added. In the figure shown, network A is nearing capacity, and a second manager is placed near the latest drilling sites to form network B. Since initially there will only be a few motes (grey diamonds) in network B, it is desirable to move a few motes (white diamonds) from network A. Network A and B have a different Network ID.

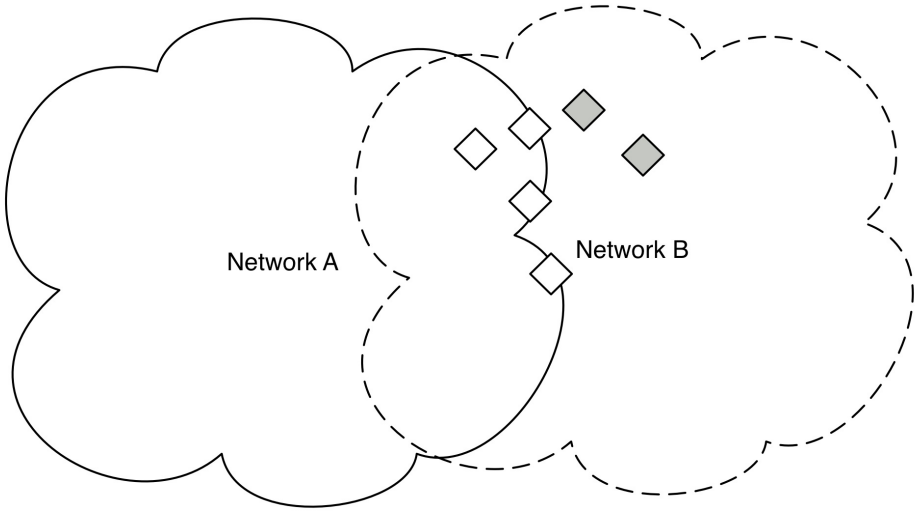


Figure 1 - Moving motes from Network A to Network B

To do this, we make use of the *search* API - this will put the mote into a mode where it listens for advertisements from any network in the vicinity and reports the networks whose advertisements it has heard.

1. The mote is preconfigured with a Network ID - we will call this the "preferred" ID. This is a normal step in commissioning a mote, regardless of whether it is ever expected to move.
2. The sensor application uses the search API to cause the mote to begin reporting information about advertisements heard - Mote ID, Network ID, and signal level (RSSI) are reported. These should be stored by the sensor application.

3. The sensor application sets a timeout for search - this should scale inversely with the *joinDutyCycle* parameter. At 10% duty cycle it should be 2-3 minutes. The longer the search at a given duty cycle, the higher the chance of hearing all networks in the vicinity but also the higher the energy used by the process.
4. At the end of the timeout, if the preferred network (the mote's current Network ID) has been heard at sufficient RSSI (> -85 dBm), the sensor application should issue a *join* command. If the preferred ID is not heard at sufficient RSSI, the sensor application should select another heard network with sufficient RSSI, set the *networkId* parameter to that network, and issue a join command.
5. If the mote tries to join the preferred ID several times but never succeeds, the mote might not have the proper credentials (see Security below) and the sensor application should restart the search process while "blacklisting" this network it cannot join.

As a general practice, this allows motes to join any available network for which they have proper security credentials. The search API can be invoked as a normal part of the joining logic used by the customer sensor application (i.e. whatever is driving the mote API), or in response to an customer application message. In the case of network A & B above, the host-side customer application would change the preferred Network ID on the white diamond motes to B, then reset those motes. Those motes will then use the procedure above to try to join network B, but can safely fall back to network A if they are not actually in range of network B.

24.3 Security

In addition to hearing an advertisement, a mote must have a join key that matches that used by the manager. Preferred practice is for each mote to have a unique join key, which is placed on the manager's Access Control List (ACL). Only motes on the manager's ACL are allowed to join the network. In the case of network A & B, the white motes being moved from network A would need to be added to network B's ACL prior to resetting them.

In a SmartMesh IP network, the manager is only capable of storing *maxMotes* ACL entries (varies by manager either 32 or 100). Examine the example of 3 overlapping networks A, B, and C. In theory this system should support 3x *maxMotes* total motes. But if we want to place a mote and have it join any one of the 3 managers, then it needs to be on all 3 ACLs. Once we want to add mote number *maxMotes*+1, we cannot, as all the ACLs are full.

One solution is to remove active motes from the ACLs of the managers they are not attached to, under the presumption that if they could exist happily in one network, they don't need to later join another. The other option is to use a common join key across all networks without an ACL - this is slightly less secure, since if the (secret) common key is known by an attacker, they can inject a mote into the network.

25 Application Note: Configuring a Network for Bounded Data Latency

25.1 Increasing Provisioning

With default settings, SmartMesh networks are designed for low-power, high-reliability operation. For typical networks with mean stability over 70% and four or fewer hops, we find that the default settings result in about 90% of packets being delivered within one reporting interval. For example, in a network where motes are each delivering one packet every 10 seconds from their sensors, we expect 90% of these packets to have latency of 10 seconds or less. By adjusting a setting called the *provisioning factor*, we can increase the power used in the network to deliver a higher percentage of packets faster than a given latency target.

In this document, we will be doing the opposite of what is described in the application note "Changing Provisioning Factor to Increase Manager Throughput."

25.2 Restrictions

The settings described in this App Note apply for networks where sensors report at roughly the same rate. For example, in a network where one mote's sensor reports at a 1 second interval and all others report at a 30 second interval, these settings will not necessarily work. For this type of scenario, the Upstream Backbone should be used. For IP networks, see the application note "Using the Powered Backbone to Improve Latency ." For WirelessHART networks, this feature is available in Manager >= 4.1.x.
Networks with low mean stability (below 70%) will have larger latency. In these cases, the provisioning should be increased even higher than the recommended settings in this document. Finally, the settings in this document apply to networks of four or fewer hops. Deeper networks have latency that increases sub-linearly, so a network of eight hops, for example, would meet the same latency targets by doubling the provisioning levels described in this document.

25.3 Provisioning

The default provisioning in all SmartMesh networks is 3x. This means that all motes have at least three upstream links for each expected packet transmission. We have found empirically that this level of provisioning provides the right balance of low power (achieved with fewer links) and high reliability (achieved with more links) for the whole spectrum of networks. So if a mote is transmitting one packet per second, including both locally generated and forwarded packets, this mote will be given three transmit links per second.

There is also a minimum number of links that each mote gets to allow it to reliably keep synchronized to its parents. For some motes, this minimum number is actually larger than the number of links that they would need to transmit data packets alone. For example, each mote in a SmartMesh WirelessHART network gets at least 4 transmit links per 10.24 seconds. If a mote has a one packet per 60 second data requirement, provisioning at 3x means the mote needs one transmit link every 20 seconds. The minimum link number of 4/10.24 is much larger than the 1/20 needed.

25.4 Changing Provisioning

As a rule of thumb, 3x provisioning will get 90% of the packets to the manager "on time". This means that motes reporting at a 10 second interval will have 90% of their packets with latency under 10 seconds and motes reporting at a 7 second interval will have 90% of their packets with latency under 7 seconds. Note that not all customers desire on-time packet delivery, often the objective is simply to deliver all of the generated packets within a reasonable, but less demanding, target.

To boost the on-time delivery to 99%, change the provisioning to 6x.

- SmartMesh WirelessHART: add a line in `dcc.ini` with `TOP_LINK_OVRSUBSCR = 6.0`
- SmartMesh IP: from manager CLI, type `set config bwmult 600`

Note that the total throughput of the network in packets per second is halved when you make this change. A network that could support 25 pkt/s with the default settings can only support 12.5 pkt/s with 6x provisioning.

To boost the on-time delivery to 99.9%, change the provisioning to 9x.

- SmartMesh WirelessHART: add a line in `dcc.ini` with `TOP_LINK_OVRSUBSCR = 9.0`
- SmartMesh IP: from manager CLI, type `set config bwmult 900`

A network that could support 25 pkt/s with the default settings can only support 8.3 pkt/s with 6x provisioning.

25.5 Power Increases

Additional provisioning doesn't result in any more packets transmitted in the network, just more frequent links to transmit the same number of packets. Leaf nodes will not have any additional power requirement when provisioning is changed as additional transmit links do not come with a power cost. Busy router nodes will be adding several receive links to pair with the additional transmit links, so these nodes will be impacted the most. Again as a rule of thumb, expect busy router power to go up 10-15% when increasing to 6x provisioning and 20-30% when increasing to 9x provisioning.
For low-traffic networks (e.g. 60 second reporting intervals), increasing provisioning to 9x may not actually increase the number of links in the network or any mote's power. In these cases, the minimum number of links is already sufficient to ensure 99.9% on-time delivery.

26 Application Note: Network Coexistence

26.1 Overlapping Networks

All SmartMesh networks operate in the same 2.4 GHz instrumentation, scientific, and medical (ISM) unlicensed wireless band. Here unlicensed means that anyone is free to operate in this band provided that they follow local power limits, but they don't need a license to operate, as they would with TV or cellular phone bands. This band was chosen for its availability worldwide and because there is a IEEE 802.15.4 PHY standard radio specification - thus there are many interoperable radios available. But because anyone can use it, it is crowded with other products - 802.15.4 b/g/n Wi-Fi, Bluetooth, Cordless phones, and ZigBee all operate in the same band. SmartMesh networks are designed with sufficient margin to transmit around potential interference by hopping through channels, under the observation that in real deployments an interferer (or multipath fade) present at channel X at time T0 may not be present on channel Y at time T1. By relying on the average performance over all channels, we see better results than relying on any one channel to be good. Since the other protocols remain on a single channel (or small set of channels), or hop quickly across the entire band (bluetooth), other SmartMesh networks in the vicinity have the potential to be the most deleterious interferers, since collisions can persist when unsychronized networks share a common schedule. This document discusses the features implemented to overcome interference from overlapping networks.

26.2 Collisions in a Single Network

SmartMesh managers schedule activity in such a way to avoid any collisions between transmissions in a single network. The exception to this rule is for "shared" links which appear in three places. Shared links are used for joining motes to contact their join parents. At worst, collisions during this time result in a mote resetting before it becomes operational in the network. In SmartMesh IP, the powered backbone uses shared links to reduce latency. These networks still have sufficient dedicated (i.e. not shared) bandwidth to satisfy all data requirements, so collisions only result in a smaller latency reduction in this case. In SmartMesh WirelessHART networks, motes share the link for sending network discovery packets. The manager assigns timers long enough to reduce the chance of these packets colliding, if they do collide the discovery information is collected safely at a later time.

None of the above specified collision scenarios collide with any regular network traffic as they are scheduled on dedicated channel offsets, and in no other case will there be two transmissions on the same channel at the same time in a single network.

26.3 Avoiding Periodic Collisions

Suppose two networks are located in the same radio space and that both networks have:

- The same current Absolute Slot Number (ASN)
- The same channel list (e.g. default channel blacklist)

- The same lengths for all slotframes
- A transaction in the same timeslot with the same offset

In this case, both networks will have a transmission at the same time at the same channel frequency and, potentially, both messages will interfere with each other resulting in neither succeeding. The packets will need to be retried at a later time. The same scenario will repeat exactly one slotframe later, and it will continue to repeat. The key is that both networks are operating with the same periodicity so a collision that occurs once is likely to occur persistently.

The networks do not necessarily even have to have matching ASN to interfere with each other if they follow the same channel hopping pattern. Networks that are not explicitly synchronized end up drifting and we observe periods where some collisions overlap. These periods, representing the time that the relative drift takes to pass through the length of a timeslot, can last from minutes to hours. If all transmissions for a mote match the pattern of transmissions in an overlapping network, this mote can end up unable to communicate for long enough to lose contact with its parents and reset. If the occasional transmission succeeds, the mote will stay in the network - it is only when all data and keep alive packets are persistently clobbered that the mote will be lost.

The amount of overlapping between two networks is very low. Each network collects data through a single Access Point (AP) node, and each AP can only listen or transmit on a single channel in each timeslot. Compared to the 15 channels that are available to SmartMesh networks, we expect a maximum of 1/15th of the overall timeslots to contain actual transmissions at the busiest location in the network. Furthermore, the transmissions don't fill the entire length of the timeslot so the fraction of the total bandwidth occupied is even lower. This suggests that there is plenty of room for motes to safely coexist in the same radio space providing we can avoid periodic and persistent collisions. If collisions occur, but happen randomly instead of consistently, this manifests as an overall decrease in path stability in both networks which doesn't cause any serious problems.

SmartMesh networks avoid periodic collisions differently in the IP and WirelessHART product families.

26.3.1 SmartMesh WirelessHART

Each SmartMesh WirelessHART network has the same slotframes with fixed lengths. The upstream slotframe is 1024 slots, the downstream slotframe is 256 slots, and the advertisement slotframe is 128 slots. Changing these slotframe lengths is not allowed, so we need a different mechanism to avoid periodicity.

Upstream communication is randomized by giving each mote a minimum of four transmit links. The offsets for these links are chosen completely randomly and there is a little jitter in the timeslot chosen for each upstream link. The chance that any four links from network A collide with network B is thus less than 1 in 15^3 ,i.e. 1: 3375.

Downstream communication is randomized by giving the AP both broadcast and multicast links and by using two source routes for each mote target. If the first attempt to reach a mote downstream fails, it will be retried with the first hop transmission on a different timeslot and relatively random timeslot, and the same holds for each subsequent transmission along the multihop route.

Advertisement and network discovery transmissions happen on a timer instead of in every assigned slot. This means that these transmissions will never occur with perfect periodicity and shouldn't cause persistent problems in neighboring networks.

26.3.2 SmartMesh IP

SmartMesh IP network managers are memory-constrained so we do not have the ability to assign several links to each mote. To overcome this limitation, we assign fewer links in shorter slotframes which does increase the chance of persistent collisions. However, since the IP networks have a single base slotframe length, this is easy to randomize. At network boot-up, the manager chooses a random slotframe length between 256 and 284 timeslots which it uses for all upstream, downstream, and advertising activity.

The default minimum number of upstream links for an IP network is two. If there aren't many motes in the network and power isn't a huge concern, this parameter can be increased in the "ini" settings to provide more immunity against persistent collisions.

Downstream communication is randomized by giving the AP both broadcast and multicast links and by using two source routes for each mote target. If the first attempt to reach a mote downstream fails, it will be retried with the first hop transmission on a different timeslot and relatively random timeslot, and the same holds for each subsequent transmission along the multihop route

26.3.3 Mixed IP - WirelessHART Environments

The IP and WirelessHART solutions operate with different timeslot lengths, 7.25 ms and 10 ms, respectively. This alone makes them safe to operate in the same radio space without fear of persistent collisions between them.

26.4 Clear Channel Assessment

All SmartMesh products feature optional Clear Channel Assessment (CCA). When enabled, all devices in the network will listen for a short period before transmitting, and abort the scheduled transmission if another device's transmission is overheard. This is intended for coexistence between neighboring unsynchronized networks - it has no effect in-network since we avoid collisions locally as described above. Use of CCA is off by default - the threshold for aborting transmit as defined in IEEE 802.15.4 is low, and networks that could operate successfully may be completely blocked by a moderate strength wide band interferer. We recommend that you only consider using CCA at the request of another nearby 802.15.4 network operator, and then only if they are experiencing coexistence problems.

26.5 Empirical Results

In our test lab, we routinely have about 40 SmartMesh networks with about 900 motes operating continuously. During times of peak traffic, we have captured upwards of 20 packet/s on each available channel in our building. This environment should be more challenging than anything faced by customer deployments and we rarely see motes reset due to collisions.

We ran a dedicated test with 1000 motes operating in our small lab split into 8x125-mote WirelessHART networks. At its peak, the total traffic was 181 packet/s in this network and the networks maintained better than 99.9% reliability. Compared to a low-traffic setting in which the motes kept synch but didn't send data, the path stability dropped from 80% to 68%. Overall, this results in increase latency and power for the network, but didn't result in any serious problems.

27 Application Note: How to Choose a Join Duty Cycle

This document walks through the decision for how to set the *join duty cycle* on the mote. This is normally done by the sensor application that talks to the mote.

27.1 Background - What is the Join Duty Cycle?

When you send the mote *join* API command, the mote starts searching for a network to join. Searching means it listens on a single channel for a while, and then sleeps for a while, and then resumes listening on a different channel. The total period (listen time + sleep time) is 3-4 seconds long (depending upon SmartMesh family). The join duty cycle is a one byte field that can be set to anything between 0 and 255, to set what portion of this period is spent listening. The greater portion of the time your mote listens the faster that mote will hear an advertisement and get joined to the network, but at increased average current consumption. For example, if you set the search duty cycle to 255 on a mote, it will listen constantly, changing channels every few seconds. If you set the join duty cycle to 26, it will spend about 10% of the time (26/255) listening and 90% of the time sleeping. This will consume much less power, but will hear an advertisement much more slowly. Thus join duty cycle setting gives you a tradeoff between average power and time spent searching. The expected time for a mote to synchronize to the network is a function of join duty cycle and network topology via the number and rate of advertising neighbors:

sync time = advertising rate per device * # channels / (# of devices advertising * path stability * listener join duty cycle)

In the SmartMesh starter kits (DC9000 and DC9007), the motes operate in **master** mode. They join on their own, with no commands from an external processor. This mode is typically used for demonstrations. In contrast, most real applications operate the mote in **slave** mode, where the mote needs to be told to join. Along with the join command the mote should be told what duty cycle it should use for searching - this document articulates the tradeoffs between "fast enough" network formation time and average power in a number of network scenarios. In SmartMesh WirelessHART the mote returns to a default when reset or power cycled, so if something other than the default value is desired, it must be set at each boot. In SmartMesh IP, the join duty cycle persists through reset and power cycle and only needs to be set once.

27.2 What Join Duty Cycle Should I Use?

There are four common approaches to setting the join duty cycle:

1. For powered devices: just set it to 100%. The benefit to this is that this is always the fastest to join. If you place a mote in a position where it cannot hear any neighbors the device will burn ~5 mA searching and never hearing any advertisements.

2. For scavenger devices: set it to the power level you can afford. A scavenger device can only guarantee a limited amount of current. Set the search duty cycle to fit in that budget. For example, if you have a scavenger that can source 200 µA continuously to the mote, and the mote consumes 5 mA in receive, then we know we have to set the search duty cycle to 10 (10/255 = 4%) or less. The mote will join more slowly than it would at a higher duty cycle, but it will stay under the power budget for as long as it needs to search.
3. Set it to match your battery life targets. If the device has a strict battery life target, then it has a specified average current budget, and you can calculate a join duty cycle as in #2 above. E.g. If you have a 2000 mAhr battery, and the target lifetime is 10,000 hrs, then you can afford 200 µA. That way the device will meet the battery life target whether it is in the network or not. This is also the approach for devices that will commonly be left in a "stranded" position but is expected to join whenever a network becomes present.
4. Try to pick a value that is "good enough" in speed and in power. If you were to test for yourself, you would find that the total time it takes a dense network (one where most motes can hear 8 or more neighbors) to fully form is only weakly associated with joined duty cycle. If it takes 30 minutes for a 100 mote network to form at 100% search duty cycle, it may only take 32 minutes at 50% and 35 minutes at 25%. By setting the search duty cycle to a lower value, you are only a little bit slower, and you save a lot of power on the "stranded" device. If you go too low, well under 10%, then things can slow down a great deal. In the limit, a search duty cycle of 0 (0.2%) will join very very slowly and will consume < 10 µA average current. If your device is not powered and is not a scavenger, and is not likely to be left stranded, so 25% is an excellent place to start. Confirm things are fast enough for your installers and consider adjusting this value based on that feedback.

27.3 The Sensor Application State Machine

SmartMesh IP motes persist the join duty cycle setting, so normally this would be set in manufacturing or when initially commissioning the device.

For SmartMesh WirelessHART, since join duty cycle is not persisted, you should consider the following rules for your application:

Rule #1: Your software should always be ready to get the Boot Event.

This event is sent by the mote whenever the mote resets. If it is power cycled, it sends the boot event. If the mote becomes lost from the network, it sends the boot event. If the mote receives a reset command over the air, it sends a boot event. The simplest state machine is to set the join duty cycle each time a boot event is received.

Rule #2: Your software should always be prepared to set the join duty cycle before it sends the join command.

Even if it turns out that you like the default join duty cycle, that default may change someday. You should always write the value you want to use (best practice is to read the current value and write a new one if different from the desired value).

27.4 Summary

To repeat, the process will be:

- Step 1: Set the join duty cycle using the mote API command in your start up routine. Make sure you call this command anytime you are going to send the join command.
- Step 2: As a placeholder, use the value 64 (25%).
- Step 3: If you don't mind consuming ~5 mA constantly, set it to 255 (100%) instead.
- Step 4: If your device is on a strict power budget, use that to calculate what join duty cycle you can afford.
- Step 5: Test your value empirically and decide if it needs to be tuned.

28 Application Note: SmartMesh Security

28.1 Introduction

Wireless packets are transmitted through the air where anyone can theoretically eavesdrop. Indeed, there are so-called *packet sniffers* that can listen to all 16 channels in the 802.15.4 spectrum (the 2.4 GHz ISM band) at the same time, so channel hopping alone is not sufficient to protect data from outside listeners. Security protocols must be designed so that a listener hearing the raw bits of every single packet still cannot decrypt any of the information. All of our products have the same security features as part of the standard product. We believe that all customers need secure networks whether or not they recognize it, and we encourage all customers to enable security. The only downside to security is that a small amount of overhead is added to each packet transmission, but the incremental power consumption this incurs is trivial. The security standards implemented in Dust products are industry best practices, and while Dust's implementation is thorough, it is not novel (that's good!).

28.2 Goals

A secure wireless network must have the following properties:

- **Message Integrity**—Data received at the destination should not be accepted if it has been modified in transit. This is an end-to-end property that must be maintained even in the presence of a malicious router and even when the packet goes through many hops from source to destination. This is also called *Authentication*, as it is intended to confirm the identity of the sender to prevent *Man-in-the-Middle* attacks where each side in a conversation is unknowingly talking to a third party.
- **Access Control**—Motes should only accept data from authorized motes. This is an end-to-end property, though it also has a link-layer corollary. Data from unauthorized motes should not be permitted to result in a denial of service (DoS) attack.
- **Confidentiality**—An eavesdropper that intercepts any encrypted data should not be able to determine anything about the plaintext data except the plaintext length (semantic security).
- **Replay Protection**—If an adversary captures legitimate encrypted traffic and re-injects it into the network (possibly at a different location), that traffic must not be accepted at the destination without detection. This is an end-to-end and a link layer property.
- **DoS resistance**—It should be difficult to inject packets into the network, congesting it to a point that prevents the network from operating normally.

28.3 SmartMesh Security Features

- **Message Integrity**—Message integrity is achieved with two 32-bit Message Integrity Codes (MIC), one at the link layer, *i.e.* at each hop, and one at the network/mesh layer, *i.e.* end-to-end within the mesh. The link layer MIC enables the receiving mote at each hop to confirm that the packet is being transmitted by a manager-approved mote. The end-to-end MIC is used at the destination to both decrypt and authenticate the packet. This MIC guarantees that the packet was not altered at any hop after the sender transmitted it. Both MICs are calculated via the CCM* algorithm (see below).
- **Access Control**—The manager maintains an Access Control List with a list of devices allowed to join the network, along with their join keys. A device cannot join without presenting the correct 128-bit Join key.
- **Confidentiality**— Confidentiality is guaranteed with a CCM* stream cipher based on AES 128-bit encryption of the payload. Encryption keys are shared secret Session Keys that are randomly generated and securely delivered at runtime. No eavesdropper outside the network can decrypt any payload. No mote inside the network can decrypt any other mote's payload.
- **Replay Protection**—In joining the network, each mote encrypts the first message using a persistent join counter – any replay join requests are detected and dropped by the manager. After the mote joins the network and obtains a session key, all packets are encrypted using a monotonically increasing 32-bit nonce counter. At the destination, the receiver tracks a history of received nonce counters to eliminate old and duplicate packets. In addition, link-layer MICs described above are computed using a timestamp based nonce. Replays are detected and dropped.
- **DoS resistance**—The security attributes that achieve Replay Protection also provide DoS resistance.

28.4 Encryption and Authentication

All data link layer (DLL) messages are authenticated with CCM* with AES-128 in hardware. This is done with join requests using a well known key, and with all other messages using the run-time Network Key. The DLL nonce counter is based on shared time (ASN) to prevent replays. Network Layer headers are authenticated and payloads authenticated/encrypted using CCM* with AES-128. The join messages use the appropriate symmetric Join Key as described above. Motes persist their nonce counter through reset. Run-time Session Keys as described above are used for all other end to end session traffic. Each session carries its own nonce, which mitigates against replay and man-in-the-middle attacks.

Message integrity and replay protection are ensured through time-based message authentication at the link layer. Session based authentication and encryption at the network layer ensure confidentiality, source authentication, and replay protection.

28.4.1 Keying Model

In SmartMesh networks, each mote contains a 128-bit Join Key stored securely in non-volatile (NV) flash. The mote authenticates itself to the network by sending a join request encrypted with this Join Key and Join Counter. The network manager decrypts this message to confirm it was sent by a mote possessing the correct key and with the expected join counter. If authentication fails, the join request is dropped.

After device authentication, the manager distributes the following Session Keys

- **Manager Unicast Session**—This session is used to securely transport all keys, and most network configuration messages. The mote communicates with other motes on a specific schedule. All commands associated with the creation and maintenance of that schedule are strictly restricted to the manager unicast session (*i.e.* only the manager can lay out network resources).
- **Manager Broadcast Session**—This session uses a single key for all motes and is used for commands that are destined for all motes simultaneously. This session is used for over-the-air-programming (OTAP) and controlling advertising rates in the network. Note that an OTAP image can only be activated using the manager Unicast Session.
- **Gateway Unicast Session**—This session is used to encrypt and decrypt sensor payloads. The manager and mote perform that encryption and decryption function.
- **Gateway Broadcast Session**—If the gateway ever needs to broadcast a payload to all sensor processors, this key will be used to encrypt and decrypt that payload.

Each of the above security sessions has a key. So in an N-mote network, there are N manager unicast session keys, and N gateway session keys, plus the two broadcast session keys. The final key is the *network MIC key*. This shared key is used by all motes to authenticate packets at the Data Link Layer. In other words, a routing mote uses this key to confirm the packet is from a valid neighbor, without possessing the key needed to decrypt that packet.

28.4.2 Manager Security Policies for Joining

The manager has three security policies for joining that can be chosen by the network administrator.

- **Accept Common Key**—In this security policy, the manager will grant network access to *any* mote that presents a network wide shared *join key*.
- **ACL with Common Key**—In this security policy, the manager will grant network access only to motes whose globally unique 8-byte MAC address are on the access control list (ACL) *and* present the network wide shared join key. Join requests that present an invalid MAC address or the wrong joinkey will be dropped.
- **ACL with Unique Keys**—In this mode, each mote on the ACL has a unique join key. A device will be granted access to the network only if it presents the correct MAC address and join key. All other requests will be dropped. **This is the recommended and most secure mode of operation.**

28.4.3 Key Management

Key generation and distribution is executed by the network manager. Key generation is based upon NIST specification SP800-90. CTR-DRBG samples thermal noise and an XOR function to generate a large random seed. There is no automatic key rotation policy in the network manager, but the network manager does serve up key rotation APIs to invoke the secure generation and rotation of any keys.

28.4.4 A Note on CCM*

CCM* stands for **C**ounter mode with a **C**BC-**M**AC. Wasn't that helpful? CBC means a **C**hain **B**lock **C**ipher – AES is a block cipher – it works on a 16-byte block of data. To make is useful for arbitrary length longer pieces of data (a *stream*), we use a technique called *Block Chaining* – using the output of one AES block computation as an input to the next block, so that the security of all the blocks is "chained" together. In CCM* we also generate a **M**essage **A**uthentication **C**ode, also called a Message Integrity Code or MIC that's used to verify that the data wasn't changed. The * in CCM* means that you can do an authentication operation, an encryption operation, or both. CCM uses an incrementing counter as a *nonce* – a "number used once" as an input to encryption/decryption operations. While it is not necessary to keep the nonce a secret, both sides need to know what the current nonce is (along with the key) to properly decode the packet, and the number must only be used once. For the link-layer nonce, the motes use the ASN – the count of slots elapsed in the network. At the network layer this is an incrementing message counter.

29 Application Note: Using the TestRadio Commands

29.1 The testRadio Commands

The SmartMesh WirelessHART Mote firmware contains API commands to measure the quality of a wireless link between two motes in a standalone test: *testRadioTxExt* and *testRadioRx*. These commands exercise low-level features of the radio and can only be used on motes which have not joined a network. They allow an integrator to expose radio testing commands programmatically through their application front end, as opposed to exposing the device CLI.

These commands are typically used to verify that an antenna design/manufacturing behaves as expected. This application note details how to use these commands to perform a Packet Delivery Ratio (PDR) test between two motes. This would typically be done for top-level assembly test or field debugging.

You will use the *testRadioTx* and *testRadioRx* commands to send 1000 packets from a transmitter mote to a receiver mote, and gather statistics on how many packets were received.

29.2 Setup

You need the following:

- 2 SmartMesh WirelessHART Motes or SmartMesh IP Motes which are not connected to any network
- A computer with 2 available USB ports and the SmartMesh SDK installed
- 2 USB cables

Before starting, write down the serial port numbers corresponding to the API port of the two motes connected to your computer.

29.3 Running the Experiment

- Start 2 instances of the APIExplorer application (part of the SmartMesh SDK).
- Connect both applications to the API port of your two motes.

- On the APIExplorer connected to the transmitter mote:
  - Select the *testRadioTxExt* command and populate the fields as follows:
    - **testType**: packet - This will instruct the mote to transmit IEEE802.15.4 compliant packets.
    - **chanMask**: 8000 -This indicates that you want the packet to be sent on channel 15 (`0x8000` corresponds to the bitmap `b1000000000000000`, i.e. only bit 15 is set), or 2.480 GHz.
    - **repeatCnt**: 1 - Number of times to repeat the packet sequence.
    - **txPower**: 8 - The mote will send the packets at +8 dBm conducted power.
    - **seqSize**: 1000 - Number of packets in each sequence
    - **sequenceDef**: pkLen = 125, delay = 20000. Packets are spaced by 20 ms, so sending 1000 packets will take 20 s.The mote will add two bytes of CRC at the end, resulting in a maximum-length IEEE802.15.4 packet.
  - Do not click on the **send** button yet.
- On the APIExplorer connected to the receiver mote:
  - Select the *testRadioRx* command and populate the fields as follows:
    - **channel**: 2.480 GHz. This corresponds to channel 15, the channel the transmitter is transmitting on.
    - **time**: 30. This instructs the mote to listen for packets for 30 s. The transmitter will transmit for 20 s so this provides a little buffer.
- Start the test:
  - Click on the **send** button on the receiver side. The mote starts receiving for 30 s.
  - Click on the **send** button on the transmitter side. The mote starts transmitting 1000 packets, for a total duration of 20 s.
- Wait 30 seconds and collect the results:
  - On the APIExplorer connected to the receiver mote, enter the *getParameter* command, and the *testRadioRxStats* subcommand. Press **send.**
  - The response fields contains the following fields:
    - **rxOk** is the number of packets received successfully.
    - **rxFail** is the number of packets received, but for which the CRC fails. This indicates the packet was received, but got corrupted during transmission.

29.4 Interpreting the Results

A number of physical phenomena can cause packets not to be received correctly: interference, multi-path fading, improper antenna connection, sender and receiver being too far apart, etc. In these cases, packets can be received at a low SNR, leading to either the packet being corrupted (and appearing in the **rxFail** counter), or not being received at all.

You should hence determine:

- How many packets were received correctly? This is the **rxOk** counter.
- How many packets were received, but corrupted? This is the **rxFail** counter.
- How many packet were lost? This is the difference between the number of packets sent and the sum of the **rxOk** and **rxFail** counters.

30 Application Note: Best Practices to Limit Average Current During Peak Periods

The LTC5800 is intended for low-power designs - but while normal mote operating currents of < 50 µA are typical, for short intervals (such as when the part is booted) current may be much higher. This poses challenges for designs that run off of a current-limited supply. The LTC5800 is designed to limit itself to an average current of 360 µA averaged over a 7 s window during these peak-current intervals, however certain system level considerations must be respected to meet this current target. This current level corresponds to the low-current boot mode in legacy SmartMesh WirelessHART products (DN2510), which was intended to support operation of the mote on a 4-20 mA current loop.

If not specifically highlighted, each item applies to all SmartMesh families.

- **Reset** – boot consumes ~800 µA and takes 800 ms. Repeatedly resetting the part after boot could raise the average current above target. It is our guideline that the OEM processor not reboot the part more often than every 2 s.
- **Join Duty Cycle** – The join duty cycle should be set to no more than 5% to ensure the average current meets this target.
- **UART activity (SmartMesh WirelessHART)** – UART traffic has a small impact on current at 9600 bps. Continuous looping on an API at 115 kbps will raise the average current by ~75 µA and should be avoided, but this is not a normal use case. Normal use of the part makes use of notifications instead of polling. The time pin should not be polled more often than every 2 s.
- **Persistent Parameters** – The system allows the user to change Flash-based nonvolatile parameters via the *setNVParameter*API – however, consecutive writes can exceed average current target. Our guidelines are:
    - Wait 2 s after receiving a boot notification to do a NV write
    - Consecutive NV parameter writes must be spaced at 2 s minimum
    - If NV writes are done after the boot notification, wait 2 s before issuing a join request
- **OTAP** – OTAP requires erasing the file system (to store the incoming OTAP file) and main Flash area (to replace the running firmware). In the current release (SmartMesh WirelessHART 1.0.2, SmartMesh IP 1.2.0), this is done atomically and will exceed the current target, so OTAP should be avoided with this version of code on current limited devices. OTAP erase will be duty cycled to meet the current target in a subsequent release. OTAP is an infrequent activity, and many systems are never upgraded in their lifetime.
- **Powered Backbone (SmartMesh IP, SmartMesh WirelessHART Mgr >=4.1)** – The low-current boot mode of the DN2510 will not meet the power target with this networking mode. LTC5800 motes can operate as non-routing leafs on the backbone in both IP and WirelessHART networks.
- **Severe congestion** – The manager limits mote links such that if every link was used, the current target would not be exceeded. In practice, only a fraction of these links are used, so the current release does not account for infrequent housekeeping operations such as radio retune due to a temperature change. In extremely rare cases, a severely congested mote could exceed the target if the device needs to retune its radio. Should this result in a reset, the mote will retune at boot and join normally.

- **Scratchpad use (SmartMesh WirelessHART mote >= 1.1)** – Starting with version 1.1.x mote software, a scratchpad is available for customer use. The guideline is the same as for persistent parameters - Scratchpad writes should be spaced by at least 2 s to ensure the average power target is met.

31 Application Note: Methodology For Pilot Network Evaluation

31.1 Summary

There is often a desire to deploy a pilot network, test it, and analyze to see if it will perform adequately in an environment representative of those in which it will ultimately be deployed. The closer the pilot deployment matches the real one with respect to mote/sensor locations and topology, their report rates, etc., the better the user gets a feel for how the real network is going to perform.

The SmartMesh SDK provides simple tools evaluating a network in a user specific environment. Here we outline a step-by-step approach that will help the user deploy the motes in a specific environment for testing. After deployment, the utilities provided with the SDK may be used to configure the different motes in the network to report data at the desired rates, thereby emulating the actual sensor reporting behavior in a real deployment. This document describes the methodology to gather statistics from the deployed network over a period of time, and analyze these statistics to evaluate the performance of the pilot network with respect to key parameters such as reliability, data rates, bandwidth, etc.

31.2 SmartMesh Starter Kits and SDK

The SmartMesh Starter Kits comes standard with 1 manager and 5 motes. Additional motes may be purchased in order to grow the network size, if required. The user needs to decide the appropriate product family for their use - SmartMesh IP or SmartMesh WirelessHART, based on prior discussion with a Linear FAE. All the documentation, software utilities and tools are available for download at http://www.linear.com/starterkits.
The SmartMesh IP Easy Start Guide or SmartMesh WirelessHART Easy Start Guide go over the installation of starter kit hardware and basic function of the SDK software. The rest of this application note assumes that the appropriate SDK has been properly installed on the user PC and the user is able to work with CLI (Command Line Interface) on the manager.

More detailed installation instructions can be found in the SmartMesh IP Tools Guide or SmartMesh WirelessHART Tools Guide.

31.3 Evaluation Methodology

These the basic 5 steps for evaluation a pilot network-

- Step 1 - SDK Installation on a PC
- Step 2 - Pilot Network Deployment of the SDK Motes and Manager

- Step 3 - Configuring Motes for Specific Data Rates
- Step 4 - Gathering Data and Statistics
- Step 5 - Analyzing Data

31.4 Step 1: SDK Installation on a PC

1. Download the SDK from http://www.linear.com/starterkits. Also download appropriate documentation based on the desired product family - SmartMesh IP or SmartMesh WirelessHART.
2. Review the Easy Start Guide document and install the SDK on your PC.
3. Make sure that you can connect the PC to the manager and exercise the CLI.
4. For SmartMesh IP make sure that Serial Multiplexer is installed and running properly. Detailed installation instructions can be found in the SmartMesh IP Tools Guide.

31.5 Step 2: Pilot Network Deployment of the SDK Motes and Manager

- Review the application note "Planning A Deployment". The key deployment guideline is that each mote be placed within range of 3 other neighbors/motes.
- Determine range based on the physical environment. You may start with a range of 50 meters and later adjust it based on your results.

For the analysis of your network it will be extremely helpful if you are able to document the locations of each mote you place, or at least the distance to their neighbors. This distance data will be crucial for making plots against RSSI once you have run your network for a while.

- Based on the above 3-neighbor guideline and the area you want to cover, determine the number of motes you will require for a pilot network deployment. Make sure you have the required number of motes for a successful deployment.
- Deploy the manager first. Make sure you have access to it via your PC. Turn it ON and make sure you can exercise the CLI commands.
- Make sure that the motes have fresh batteries. They ship with fresh batteries but during install and testing of the SDK, they may have been drained. This can happen if the mote is left on without the manager/network being available. In this case the mote keeps searching for a network with the receiver on 25% of the time which might deplete the capacity.
- Start by placing the motes where you are likely to place the sensor for monitoring specific parameter. Turn the power switch ON. For SmartMesh IP motes, if enabled, the Status_0 LED will begin blinking.
- Note the MAC Address of the mote and the location it is deployed appropriately in a map or a table as desired. This will be helpful when you will configure the mote for data rate, in Step 3.

⚠ Before placing the motes, make sure they are configured in **master** mode. This is their default setting, but during testing of the SDK you may have changed their configuration.

- Follow the methodology outlined in the planning Application Note to place additional motes, if required, which will act as repeaters.
- For SmartMesh IP, once all motes are deployed, the two Status LEDs will be lit, indicating that each mote is now connected to the manager.
- Now go back to the manager to check that the network has formed. You can do this via the manager CLI connection. Use the command `sm` (short for "show motes") to display a list of the motes in the network.

⚠ It can take several minutes for the network to form once deployed, depending on the size of the network. If a mote doesn't seem to join, try restarting it. You can also trace mote state changes with CLI command `trace motest on`. This will post notifications as the motes step through the join process.

Figure 1 below shows five motes joined to a manager, as displayed with `sm`. Mote ID 1 is the AP (Access Point) mote. Make sure all the motes in your network show up on this list.

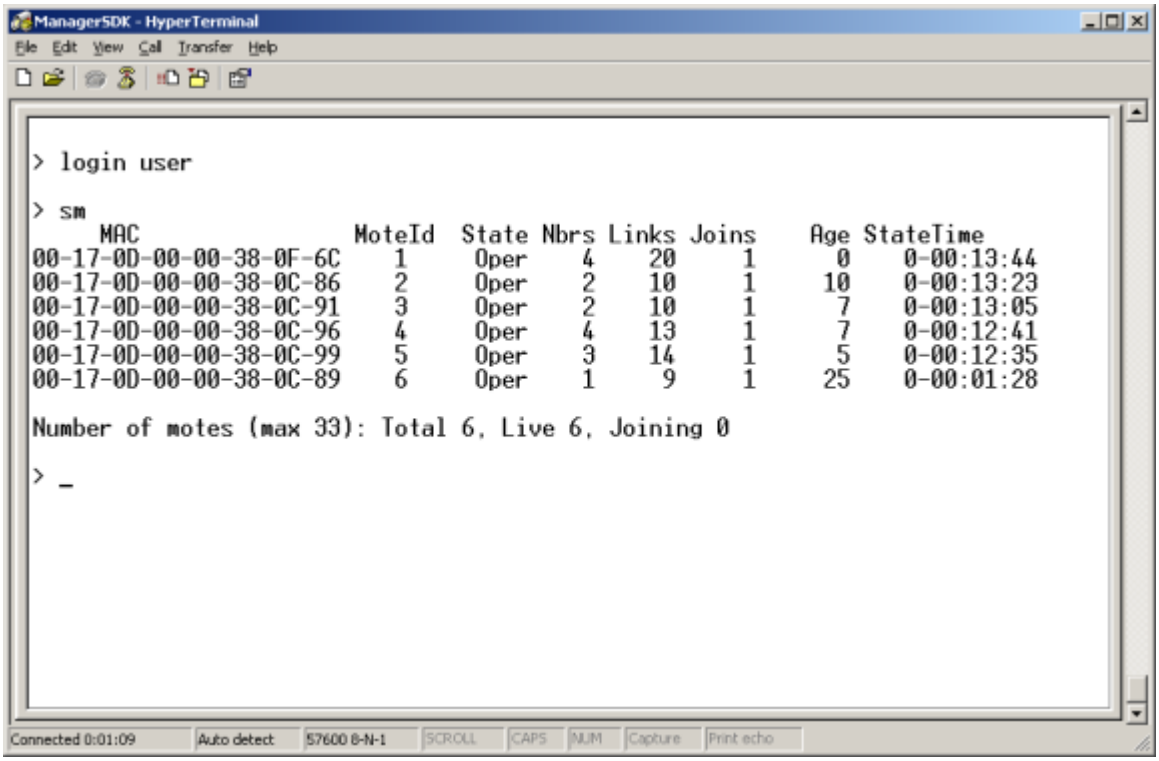


Figure 1 – CLI command `sm` showing the motes connected to the manager

31.6 Step 3: Configuring Motes for Specific Data Rates

Now that the network is up and running, we will need to configure the motes to generate and send data as if they were collecting it from sensors. We will specify the quantity, rate and size of the data we will send upstream (from mote to manager) through the network. We will use the PkGen utility available with SDK to accomplish this.

The packet generation utility PkGen allows for configuration of each mote in the network from the manger itself. This means that the user does not have to physically go to individual motes to set their rate. Also, these configurations can be changed as desired, simply by entering new fields for the mote that needs the change. Since a SmartMesh network can operate as a heterogeneous network, each mote may be configured for its own data rate: you can have a combination of slow/medium/fast data rate motes, emulating the real world scenario of slow/medium/fast sensors.

Here are the steps to follow for configuring the motes using the PkGen utility:


For this section you will need to have the serialMux running

- Navigate to the `/win/` directory in the SmartMesh SDK, and launch PkGen by double-clicking on the Windows executable at `/win/PkGen.exe`
- It will give you a prompt where you choose between SmartMesh IP or WirelessHART. Pick the appropriate family and click **load**.
- If you are connecting to an IP device, just click connect through serialMux, when prompted. If this is not functional, make sure that the serialMux is installed properly in Step 1.
- If you are connecting to a WirelessHart device, enter the IP address of your manager when prompted. The default static IP address of the WirelessHART manager is 198.162.99.101. You can change it to any address desired. Please refer to the SmartMesh WirelessHART User's Guide for details.
- When you connect, the fields you entered should turn green and the list of motes in your network should populate the mote list section. Each mote in your network will have a series of fields in the table that you will enter to configure each mote as desired.
- The **mac** column gives the MAC Address of the mote connected to the network, which is to be configured. You may use this to determine the location of the mote in the field and then determine specific configuration.
- The next column, **num. pkgen**, is a counter of the packets sent to the manager from that mote at any given time once you start PkGen.
- The next column, **pk./sec**, is another counter that shows the average packets per second rate at which the mote is sending packets to the manager. It may take awhile for this to show up.
- The previous two columns can be reset by pressing the **clear** button. You will likely want to do this every time before you set new send parameters.
- The last column has three fields and the **set** button. The first field is for the number of packets total the mote will send. Depending on the duration of the deployment and the rate at which the mote is going to send data, enter the appropriate number here. You must send at least 1 packet, but there is no max. The second field is the interval between packets in milliseconds. (*e.g.* 1000 = 1 packet/ 1000ms or 1 packet per second). The final field is the size of the packet you want to send each time. (max 60 bytes with a 20 byte header).

⚠ Do not fill in the blank cells to be filled by the user with a "-". Leave them blank.

- Click on the **set** button on the last column to start the motes sending packets. (The **clear pkgen** button allows you to reset these fields, if desired.)

⚠ Wait between presses of the **set** button. It you hit **set** twice quickly, the cells may turn green erroneously indicating that they were updated, but no packets will be sent into the network.

- Repeat the above 2 steps for each of the motes in the network.

⚠ There is an upper limit to the total bandwidth available at the manager to receive packets - it is family-dependent, but ~25 packets/s. This limit should not be exceeded. In other words, the sum total of the data packets per second coming from all motes going to the manager should not exceed this limit. The data-rate at individual mote thus depends on the total number of motes in the network and their individual data rates. For example in a network with 5 mote, you should not exceed 5 packets per second on all the motes at once.

- Closing the script does not mean that the motes will stop sending packets. They will continue to do so until they have sent all packets. Reset the motes if you want them to stop them from sending data.

Figure 2 below shows an example of the PkGen screen shot. For example, If you wanted to send 1 packet per second at max size for a week, you would fill out the **num/rate/size** fields: **604800 1000 60**
Once you have set the parameters to your liking for each mote you should see the **num. pkgen** counter start counting. The **pk./sec** column will display the rate rounded to the nearest 0.1 packet per second. If the rate is too slow or not enough packets are sent (for it to calculate an average) it will display a rate of 0.0.

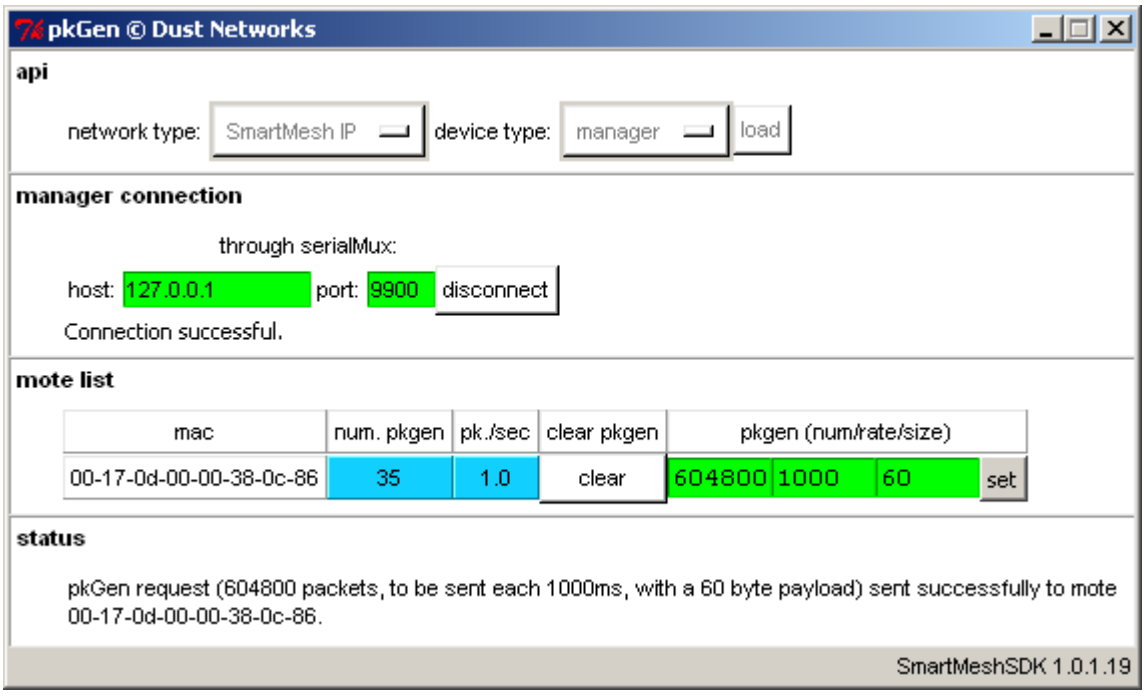


Figure 2 - PkGen screen shot showing configuration of a mote

31.7 Step 4: Gathering Statistics

Now that the network is sending data upstream, the next step is to gather network statistics that can be used to analyze the performance of the network in its current configuration. Every mote in a network periodically sends a packet which contains information about the network, packet success rate, neighbor information etc. These packets are called *health reports* and are typically sent every 15 minutes. A SmartMesh network has built in APIs that provides access to all the network statistics to the user, just as the data access.

You will take a snapshot of the statistics in one of two ways, depending on if you are using SmartMesh WirelessHART or SmartMesh IP

Step 4A: Gathering Statistics in an IP network

The SmartMesh IP manager maintains a minimal set of statistics for network reliability, path stability, and mote behavior. To get a complete picture of network health, the user must log all health report notifications. This topic is discussed in more detail in "Application Note: Monitoring SmartMesh IP Network Health."

Step 4B: Gathering Statistics with WirelessHART NetworkSnapshot

The SmartMesh WirelessHART manager contains a utility for collecting all manager logs - it is invoked by logging into the manager and using the command at the Linux prompt.

1. Log onto the manager using the dust login the way you would to connect to CLI.

2. Instead of using nwConsole you will use the command

```
/usr/bin/create-network-snapshot
```

This will take a snapshot of current network statistics and store it in `/tmp/snapshot/snapshot.tar.gz`

1. You can access this snapshot by connecting to the manager with WinSCP or another FTP client. You will connect to the same IP address you used to connect with pkgen and port 22. Navigate to `/tmp/snapshot` and transfer `snapshot.tar.gz` to your machine.

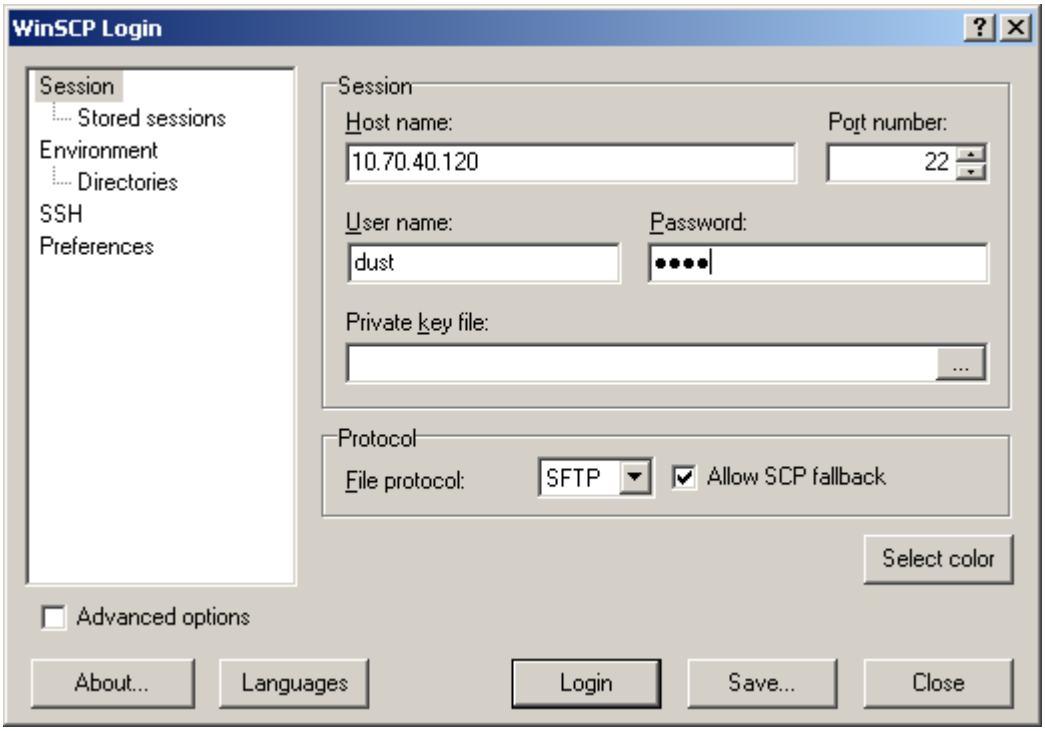


Figure 3 - Transferring Network Snapshot data from the manager

Taking a new snapshot overwrites the old one so make sure to transfer the file off of the mote before taking the new one.

31.8 Step 5: Analyzing Data

Example tools to streamline the collection of snapshot logs in SmartMesh IP, and to analyze the logs produced by the snapshot features are still in development, but will be part of the SmartMesh SDK when available.

One goal of inspecting snapshot logs is to obtain data about path RSSI so that it can be compared to distance in order to build better networks. By finding which paths have stronger signal you will be better able to judge the optimal mote spacing.

31.8.1 Wireless Hart Snapshot Log

This log will be found in the .tar file you extracted from the manager. Once you have unzipped it the data you seek is found in the `nwconsoleOut.txt` file. This log, accordingly with its name, is in fact just the output of an internal querying of the console on the manager. It issues a variety of commands to gather information about the state of the network at the time of snapshot. The first commands, beginning with `show ver` give basic information about the version, status and settings of the manager in general.
`sm –a` will print the list of motes in the network.

Here is a segment of that output.

```
< sm -a
Current time: 07/24/12 09:15:03 ASN: 32492664
Elapsed time: 3 days, 18:15:30
MAC MoteId Age Jn UpTime Fr Nbrs Links State
00-17-0D-00-00-1B-1B-CD ap 1 1 3-18:15:23 6 10 106 Oper
00-17-0D-00-00-38-0C-86 2 16 2 19:02:46 2 2 11 Oper
00-17-0D-00-00-38-0C-89 3 9 2 1-00:03:00 2 7 17 Oper
```

The `get paths` command that follows will list every mote pairing and the number and direction of links on that path, as well as path quality. If no links exist it will be listed as unused and have a default path quality of 75.

```
< get paths
pathId: 65538
moteAMac: 00-17-0D-00-00-1B-1B-CD
moteBMac: 00-17-0D-00-00-38-0C-86
numLinks: 5
pathDirection: downstream
pathQuality: 90.07
pathId: 131077
moteAMac: 00-17-0D-00-00-38-0C-86
moteBMac: 00-17-0D-00-00-38-0C-96
numLinks: 0
pathDirection: unused
pathQuality: 75.00
```

Following this will be more commands with useful but less applicable information, but then there are the `show mote -a` commands that make up the bulk of the information you will want. These will show all the paths to neighbors for each mote as well as the RSSI and path quality for each of those paths.

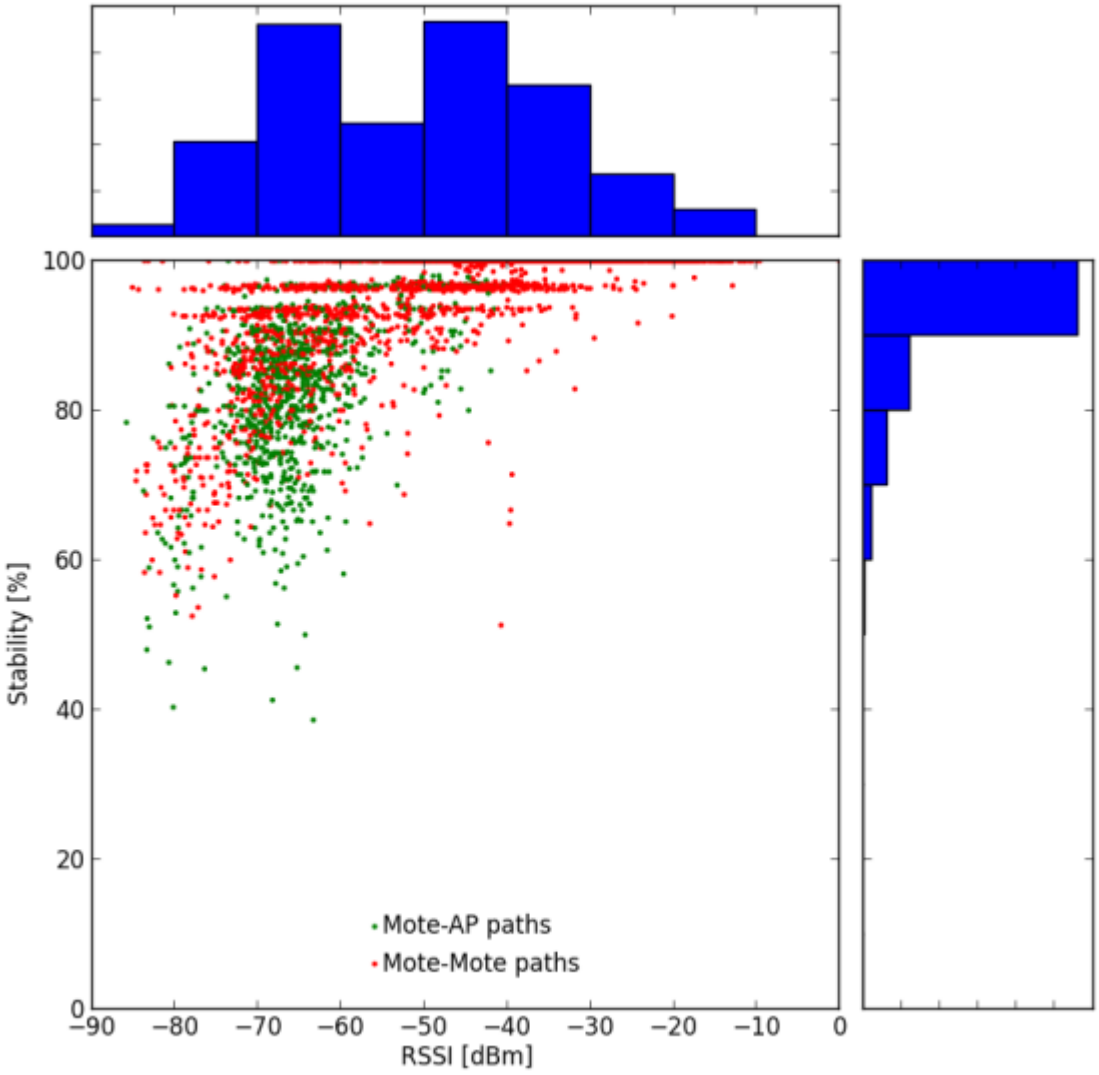
```
< show mote 1 -a
00-17-0D-00-00-1B-1B-CD 1 Oper SW: 3.0.2-0 HW: 37
Location is not supported
Number free TS: 757
Upstream hops: 0, latency: 0.000, TTL: 127
SourceRoute: Dist(Des): 0.0(10) Prim: 1 Sec:
Power Source: line
Advertisement Period: 20.000
Bandwidth:
Summary for AP: 2.5342
Neighbors: 10. Links: 106. Norm/bitmap 21/85 (max norm: 63)
Links per second: 19.824219 (unlimited)
Frame: 0. Neighbors: 10. Parents: 0. Links rx:87, tx:1.
Broadcast links
0. 1. 0: rtdb
0.993.10: rjb
<- #2 Links 3/3/3 RSSI: -52 Q: 0.90
<- #3 Links 4/4/4 RSSI: -29 Q: 0.94
<- #4 Links 8/8/8 RSSI: -50 Q: 0.83
<- #5 Links 5/5/5 RSSI: -51 Q: 0.86
<- #6 Links 3/3/3 RSSI: -48 Q: 0.90
<- #7 Links 33/33/33 RSSI: -41 Q: 0.86
<- #8 Links 16/16/16 RSSI: -43 Q: 0.90
<- #9 Links 6/6/6 RSSI: -45 Q: 0.87
<- #10 Links 4/4/4 RSSI: -41 Q: 0.90
<- #11 Links 3/3/3 RSSI: -41 Q: 0.95
```

Following this section the log will be populated with many tables of stats; daily stats for motes, paths and the network, as well as lifetime stats for the same.

31.8.2 Log File Interpretation and Analysis

The goal is to generate a waterfall plot that lists distance versus RSSI or path stability. The knee of this plot gives the user a good idea of what range is appropriate in the deployment environment. This range may be used for future deployments as opposed to the default of 50 m that was used in the original deployment.

A sample plot is shown below.



These plots can be generated by:

- Extracting the RSSI (or path quality) values for each path from the log file
- Calculating the distance between each mote pairing
- Plotting each RSSI (or path quality) value against the corresponding distance

31.8.3 Neighbor Stability Analysis

The number of good neighbors (strong path stability) a mote has is crucial to forming resilient networks. Gathering this data from the log files can help find weak links in the network mesh. This feedback would be particularly useful to someone deploying a network as motes with 3 or more good neighbors have a high probability of successfully joining the network. Those with less than 3 good neighbors will likely need more neighbors to be placed nearby.

Other network heath and performance may be obtained as described in the relevant family "Application Note: How to Evaluate Network and Device Performance."

32 Application Note: What is Packet ID and why do I Need it?

32.1 Scope

This document describes the utility of the Packet ID (bit 1) in the fields flags found in the Mote API header in both Wireless HART and SmartMesh IP Mote products. The first section of this document briefly describes the Packet ID. The second section describes how there are two Packet IDs being used across the Mote-to-Sensor Processor interface, each of which being toggled and tracked independently. The third section has some useful tips on how your code should use Packet ID and the related Sync Bit.

32.2 What is Packet ID?

Imagine we have a motorized bucket filler, and we have a communication interface where we instruct our device to:

1. Move to the right
2. Check that there is a bucket
3. Release one bucketful of water
4. Repeat

We could write a microcontroller application that tells our bucket filler to do this, and it would usually work. Imagine that we want this interface to be more robust to communication errors, so we implement a *call-response* protocol. For each command from the client (here the microcontroller), we generate a response from the server (here the bucket filler). Our transactions start to look more like this:

1. Microcontroller: Move to the right
2. Bucket filler: I moved to the right
3. Microcontroller: Check that there is a bucket
4. Bucket filler: There is a bucket
5. Microcontroller: Release one bucketful of water
6. Bucket filler: I released one bucketful of water
7. Repeat

Now we have a more robust interface, since the client gets confirmation that its message succeeded. But we can anticipate further problems when the message does not go through. If the microcontroller sends the command “Move to the right” and receives no response, it could be due to the bucket filler not getting the command, or the microcontroller not getting the response - in the former case the client needs to repeat the command, whereas in the latter it should move on to the next command to avoid confusing the server. What we need is an identifier that will allow us to tell the difference - we can do this by sending a counter along with the command:

1. Client: Move to the right (ID = 0)
2. Server: I moved to the right (ID = 0)
3. Client: Check that there is a bucket (ID = 1) - the command does not go through
4. No response received, so the client repeats the last command
5. Client: Check that there is a bucket (ID = 1)
6. Server: There is a bucket (ID = 1)
7. Client: Release one bucketful of water (ID = 2)
8. Server: I released one bucketful of water (ID = 2) - the response does not go through
9. No response received, so the client repeats the last command
10. Client: Release one bucketful of water (ID = 2)
11. Server: knows that it already filled the bucket in response to command (ID = 2), so it drops the command and replies "I released one bucketful of water (ID = 2)"

Because there was an identifier on the command, the receiver of the command can distinguish between a retry and a new command. The bucket filler knew that it didn't need to fill the bucket again, because it did not receive a different ID on the command. For *reliable* interfaces that don't advance to the next command until the current command is acknowledged, a 1-bit counter can be used for ID, since we only need to distinguish between a command and the command that follows. Command 1 follows command 0, and command 0 follows command 1. This is how the 1-bit Packet ID field works - by toggling the Packet ID bit between 0 and 1, the receiver can distinguish between a retry and a new command. Most Mote commands don't have a bad consequence like filling a bucket twice or moving two steps to the right instead of one, but it is still a major improvement to the reliability of an interface.

32.3 There are Two Packet IDs

In a device containing a Mote and a Sensor Processor, there are two independent communications streams - one where the Mote is server and the Sensor Processor is client, and another where the Sensor Processor is server, and the Mote is client. Each stream has its own Packet ID. Every time the Sensor Processor sends a command to the Mote, it must toggle its Packet ID. The Mote will check the Packet ID on every command received, comparing it to the previous Packet ID received. If the Packet ID is different than the stored value, then the command is processed and this Packet ID is stored. If the Packet ID is a repeat, the command is not processed and the Mote will reply with the same reply it sent on the previous command (the cached response).

Since the Mote can send commands (Notifications) to the Sensor Processor at any time, there is a second Packet ID for that communication. Every time the Mote sends a command, it will toggle its Packet ID. The Sensor Processor must check the Packer ID, comparing it to the previous Packet ID received. If the Packet ID is different than the stored value, then the Notification should be processed, and this Packet ID should be stored. If the Packet ID is a repeat, the Notification should be dropped and the microcontroller should send the previous cached response.

32.4 The Sync Bit

If the Sensor Processor application resets or for any reason, it no longer knows which Packet ID to use or to expect. The Sync bit allows the client to reset the Packet ID to a known state. The Sensor Processor should set the Sync Bit on the first packet it sends to the Mote - this tells the Mote that this is a new packet, regardless of the Packet ID. The Sensor Processor must toggle the Packet ID from that point forward. When the Sensor Processor receives the first packet after it resets, it should accept any Packet ID and expect the Packet ID to toggle for further packets. In SmartMesh IP, should the Mote reset, it will always send a boot event with the Sync Bit set. In SmartMesh WirelessHART, the Sync Bit will not be set, but the Sensor Processor can treat each boot event as unique, so it can accept any Packet ID from the Mote and expect the Packet ID to toggle for further packets.

32.5 Ignoring Packet ID

In order to simplify a Sensor Processor implementation to just get it working, the Sensor Processor can effectively ignore Packet ID by processing every command on the receive side assuming it is new. On the transmit side, it should set the Sync Bit on every message, ensuring that the Mote will process every packet as a new command. You will lose some of the robustness of the reliable call-response interface by doing this, so this is not the recommended behavior for anything but a prototype.

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

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.