Dust Networks

SmartMesh® IA-510
Mote Serial API Guide
Industrial

Trademarks

SmartMesh-XR, SmartMesh-XT, SmartMesh-XD, and SmartMesh IA-510 are trademarks of Dust Networks, Inc. The Dust Networks logo, Dust Networks, Dust, and SmartMesh are registered trademarks of Dust Networks, Inc. All third-party brand and product names are the trademarks of their respective owners and are used solely for informational purposes.

Copyright

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

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

Disclaimer

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

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

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

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

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

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

© Dust Networks, Inc. 2009, 2010. All Rights Reserved

Document Number: 040-0067 rev 4 SmartMesh IA-510 (I) Mote Serial API Guide

Last Revised: August 19, 2010

Document Status	Product Status	Definition
Advanced Information	Planned or under development	This document contains the design specifications for product development. Dust Networks reserves the right to change specifications in any manner without notice.
Preliminary	Engineering samples and pre-production prototypes	This document contains preliminary data; supplementary data will be published at a later time. Dust Networks reserves the right to make changes at any time without notice in order to improve design and supply the best possible product. The product is not fully qualified at this point.
No Identification Noted	Full production	This document contains the final specifications. Dust Networks reserves the right to make changes at any time without notice in order to improve design and supply the best possible product.
Obsolete	Not in production	This document contains specifications for a product that has been discontinued by Dust Networks. The document is printed for reference information only.

Contents

About This Guide

Related Documents 1
Conventions Used 1
Revision History 2

1 Introduction

Solution Overview 1
Communications between Mote and Microprocessor 2
Acknowledged Serial Link 2
Packet Retransmission 2
Mote Bring-up and Power Cycling 4
Mote State Machine 5
Configuring the Mote 6
Sending Data to and Receiving Data from the Network 7
Best-effort Communication 7
Reliable Communication 7
Downstream Reliable Retries 8
Application Support 9
Bandwidth Services 9
Non-service Bandwidth Control 14
Timestamps 14
Events and Alarms 14
RF Compliance Testing Commands 15

2 Mote Command Reference

HDLC Packet Format 17
Request and Response Packet Format 17
Command Type 18

Length 18
Flags 18
Result Codes 19
Generic Acknowledgement Packet 19
Data Types 20
Byte Ordering 20
Concatenating Command Payloads 21
Command Summary 23
Microprocessor to Mote Commands 24
setParameter 24
setParameter<txPower> 25
setParameter<joinDutyCycle> 26
setParameter<batteryLife> 27
setParameter<service> 28
setParameter<eventMask> 29
getParameter 30
getParameter<moteInfo> 31
getParameter<networkInfo> 32
getParameter<moteStatus> 33
getParameter<time> 34
getParameter<charge> 35
getParameter<testRadioRxStats> 36
getParameter<service> 37
setNVParameter 38
setNVParameter<macAddress> 39
setNVParameter<networkID> 40
setNVParameter<txPower > 40
setNVParameter<joinKey > 41
setNVParameter<powerInfo> 42
setNVParameter<OTAPlockout> 43
getNVParameter 44
getNVParameter<macAddress> 45
getNVParameter<networkID> 46
getNVParameter<txPower > 46
getNVParameter<powerInfo> 47
getNVParameter<OTAPlockout> 48
send 48
join 50
disconnect 51
reset 52
lowPowerSleep 52

testRadioTx 53
testRadioRx 54
clearNV 55
Mote to Microprocessor Commands 56
timeIndication 56
serviceIndication 57
events 59
dataReceived 60
HDLC Packet Processing Examples 61
A Changing Network ID During SEARCH State

About This Guide

This guide describes the SmartMesh® IA-510 mote serial API.

Related Documents

The following documents are available for SmartMesh-enabled networks:

- *SmartMesh Mote Serial API Guide* (this guide)
- *SmartMesh Manager Serial API Guide*
- *SmartMesh Manager XML API Guide*
- *CLI Commands Guide*
- *Product Datasheets*

Conventions Used

The following conventions are used in this guide:

- `Computer type` indicates information that you enter, such as specifying a URL.
- **Bold type** indicates buttons, fields, and menu commands.
- *Italic type* is used to introduce a new term.

Note: Notes provide more detailed information about concepts or consequences.

Caution: Cautions advise you about actions that might result in a loss of data.

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

CONFIDENTIAL

Revision History

Revision	Date	Description
040-0067 rev 1	6/12/2009	Product release.
040-0067 rev 2	8/13/2009	
040-0067 rev 3	11/20/2009	
040-0067 rev 4	8/18/2010	

Introduction

1

The SmartMesh IA-510 Mote Serial API provides OEMs with a well-defined, easy-to-integrate application programming interface (API) via a mote serial interface. Through this interface, OEMs have access to the rich capabilities of the mote and the network.

The API includes commands for configuring and controlling the mote, querying mote settings, sending and receiving data over the wireless mesh network, and testing RF functions.

Solution Overview

Dust Networks SmartMesh IA-510 is an industry leading wireless networking solution designed for critical monitoring and control applications in industrial and commercial markets. With applications ranging from renewable energy generation like utility scale solar and wind power, to factory machine health and HVAC monitoring and control. The mote serves as a wireless subsystem that may be integrated into a device. The embedded manager is designed for integration into a gateway controller, which may in turn communicate with a control program. The diagram in Figure 1 shows terminology used throughout this guide.

The mote serial API described in this guide serves as the (OEM) microprocessor interface to the mote itself and the wireless network beyond it.

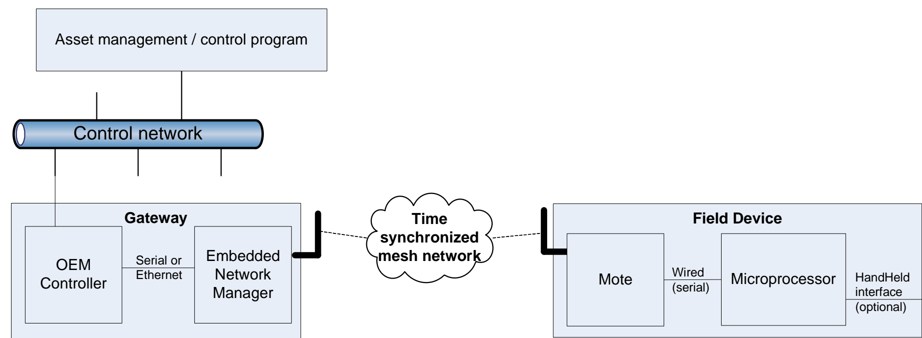


Figure 1 A SmartMesh Embedded Wireless Sensor Networking Solution

Communications between Mote and Microprocessor

The connection between the mote and microprocessor is a wired serial connection. Refer to the mote product datasheet for details and specifications on voltage signaling levels, serial handshake definition, and signal timing. The mote serial API described in this document operates over this interface.

Acknowledged Serial Link

All packets sent across the serial link must be acknowledged. The acknowledgement must be received before another serial packet may be sent. This applies to commands initiated by either the mote or the microprocessor, as shown in Figure 2.

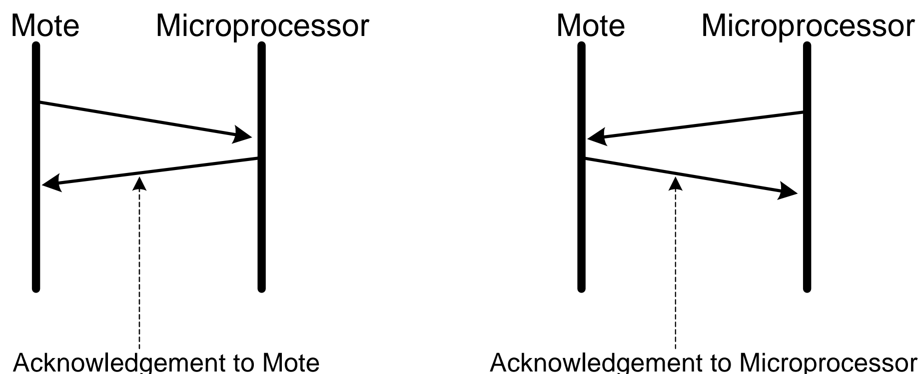


Figure 2 Communications between Mote and Microprocessor

All serial commands in the API have a corresponding response packet that serves as an acknowledgement. Generic acknowledgement packets are defined below for the mote and for the microprocessor if either receives a well-formed packet that has an unknown command type. If a packet sent from the mote to the microprocessor is not acknowledged, the mote assumes the packet has been lost and will retransmit the packet (see the following section, “Packet Retransmission”).

Packet Retransmission

The mote supports packet retransmission across the serial port for both mote-to-microprocessor and microprocessor-to-mote transmissions. Although the serial link between the mote and microprocessor is generally reliable, packet loss or bit corruption is possible. The mote or microprocessor should retransmit packets if no response is received.

When the mote sends a serial packet to the microprocessor, it expects a valid response packet in reply. If the mote does not receive a response by $t_{diag_ack_timeout}$, as defined in the mote product datasheet, the mote will continuously retransmit the unacknowledged

packet until it receives a positive acknowledgement from the microprocessor. If the OEM microprocessor cannot respond within $t_{diag_ack_timeout}$ then it must not respond at all. During the $t_{diag_ack_timeout}$, the mote is available to respond to other commands from the microprocessor.

Similarly, the microprocessor should retransmit serial packets not acknowledged by the mote. The number of times the microprocessor chooses to resend packets is application dependent, with the exception of send packets that are end-to-end responses. *Send* packets that are end-to-end responses must be retransmitted continuously or the end-to-end network session to the device will be broken.

To distinguish between new and retransmitted packets, the sender must toggle the packet ID flag bit (see “Flags” on page 18) if a new packet is sent, or leave the flag unchanged for retransmissions. If two or more packets with the same packet ID are received back-to-back, all but the first packet should be ignored and RC_OK should be returned in the response.

For example:

- The mote sends a request packet with packet ID = 1.
- The microprocessor successfully receives the packet, and sends response.
- The response becomes corrupted and fails CRC check at the mote.
- The mote times out and retransmits the original packet (still with packet ID = 1).
- The microprocessor receives the packet, and discards it because it matches the packet ID of the last request.
- The microprocessor sends the response packet with RC_OK.
- The mote successfully receives response packet. When it sends the next packet, it uses packet ID = 0.

Note that if the microprocessor resends a packet because the result code in the response packet is RC_NO_RESOURCES, the retry is considered a new packet and the sender must toggle the packet ID flag bit.

For example:

- The microprocessor sends a request packet with packet ID = 1.
- The mote successfully receives the packet, but does not have resources to process the packet and replies with RC_NO_RESOURCES.
- The microprocessor should retry sending the packet with packet ID = 0.

When the mote boots, the packet ID counter is reset and therefore the boot event message always has a packet ID of 0.

The OEM microprocessor must maintain a packet ID counter for validating packets it receives and for inclusion in packets that it sends. When replying to or acknowledging a packet, the packet ID of a response must be the same as the packet ID of the request.

- The microprocessor sends a request packet with packet ID = 1 and receives its ACK.
- The mote sends the microprocessor a request with packet ID = 1;
- The microprocessor sends the response to the mote with packet ID = 1.
- The microprocessor sends a new request packet, and uses packet ID = 0.

When used with reliable network transport, the packet ID mechanism ensures that one and only one copy of a packet sent through the network will be received by the recipient. The packet ID feature improves the reliability of the serial interface between the mote and the microprocessor and is therefore highly recommended. However, the OEM microprocessor does have the option of disabling packet ID toggling to simplify coding at the increased risk of duplicate packets due to retransmissions across the serial interface. Specifically, the OEM microprocessor may elect to ignore packet IDs during one or both of the following:

- **Requests from the microprocessor**—When sending requests to the mote, the microprocessor may disable packet ID toggling (in both the request and response) by setting the "Ignore packet ID" bit in the flags byte of the request. In this case, the mote cannot discern duplicate serial packets and thus will process all serial packets it receives.
- **Requests from mote**—The mote will always send requests to the microprocessor with packet ID toggling enabled. The OEM microprocessor should maintain an internal packet ID to discern duplicate packets from the mote. However, if the application tolerates the possibility of duplicate packets due to serial retries, the OEM microprocessor may ignore the packet ID of the incoming request. In this case, the OEM microprocessor should acknowledge the request with the "Ignore packet ID" bit in the flags byte set to ignore.

Mote Bring-up and Power Cycling

Most sensor implementations use microprocessor General Purpose IOs (GPIOs) to interface to the mote. It is important that the microprocessor firmware comply with the GPIO voltage input limits of the mote during power up. The power-on and power-off sequence described below must be followed if some of the mote inputs are powered to logic high but are not powered simultaneously with the mote. The powering sequence is not required under the following conditions:

- All mote inputs are driven to logic low.

Or

- Some mote inputs are driven to logic high, but are all powered up simultaneously with the mote.

Power-on sequence:

1. All inputs to the mote should be in tri-state, or actively driven to low.
2. All inputs should be set to low, and then enabled if they are in tri-state.
3. Power should be enabled.
4. After a delay to allow for the power supply to the mote to stabilize, IOs should be set to their default state, including the negation of reset (if a reset signal is incorporated in the design.

Power-off sequence:

1. A disconnect message should be sent to the mote.
2. If a reset signal is incorporated it should be driven low.
3. All other inputs to the mote should be driven low.
4. Power should be disabled. Note that power should be disabled for a sufficiently long period to ensure that the supply voltage has fallen to ground.

Mote State Machine

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

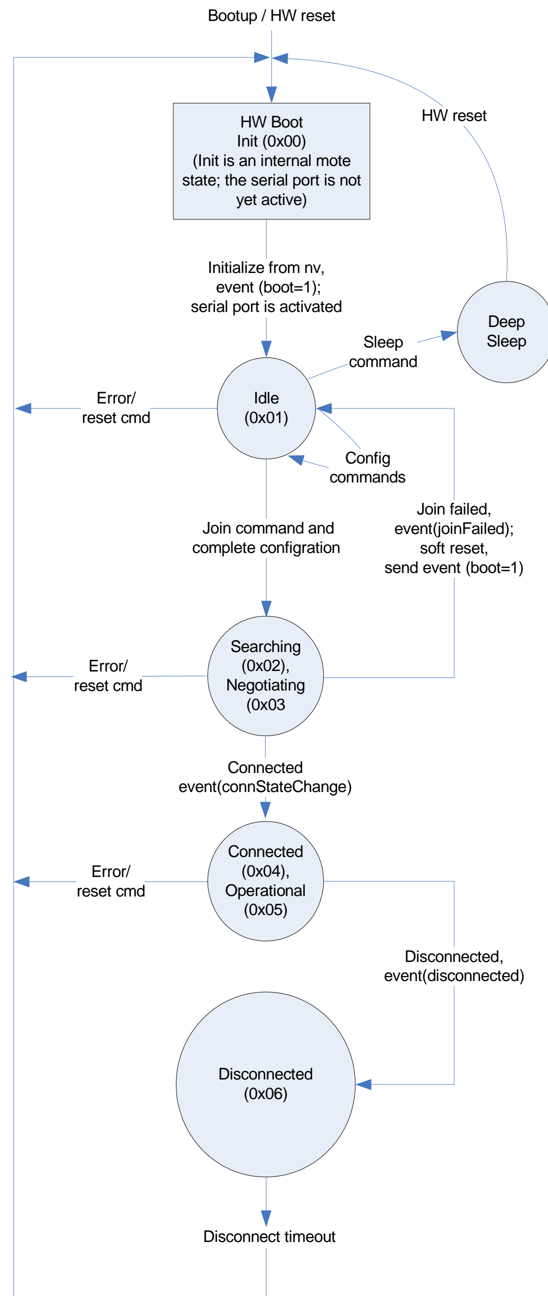
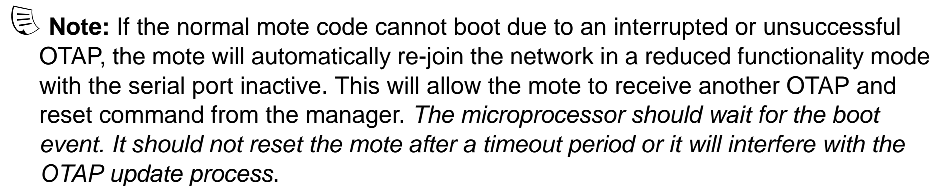


Figure 3 Mote State Machine

The mote states are as follows:

- **IDLE**—While in this state, the mote accepts configuration commands. Upon receiving a join command, the device moves into the SEARCHING state.
- **DEEP SLEEP**—The mote enters deep sleep when it receives the *lowPowerSleep* command from the attached serial processor. In this state, the device can no longer respond to serial commands and must be reset to resume normal operation. For power consumption information, refer to the mote product datasheet.
- **SEARCHING**—In this state, the device keeps its receiver on with a configurable duty cycle while searching for the network.
- **NEGOTIATING**—The mote has detected a network and has received a *join* request from the manager.
- **CONNECTED**—The mote has joined the network and established communications with the network manager, but has no links for sending or receiving application data.
- **OPERATIONAL**—The mote has links to the network manager and the device has sufficient bandwidth for basic communication with the control program (through the network manager).



Note: If the normal mote code cannot boot due to an interrupted or unsuccessful OTAP, the mote will automatically re-join the network in a reduced functionality mode with the serial port inactive. This will allow the mote to receive another OTAP and reset command from the manager. *The microprocessor should wait for the boot event. It should not reset the mote after a timeout period or it will interfere with the OTAP update process.*

Configuring the Mote

The IDLE mode provides an opportunity for the microprocessor to configure key operating parameters on the mote and control joining.

The minimum set of parameters required for joining are stored in mote’s non-volatile memory (NV), and may be set with the *setNVParameter* command. The values updated with the *setNVParameter* command are stored in NV and used after the next mote boot. Because some parameters may be safely updated without a mote reboot, the *setNVParameter* command also has an option to update the value of the parameter in mote RAM. Refer to the command descriptions in Chapter 2 for details on the effects of updating RAM. Writing to non-volatile memory causes a temporary increase in current consumption and the lifetime number of writes to flash is limited. For these reasons, the *setNVParameter* command should be used sparingly.

Note that additional parameters may be set using the *setParameter* command. The mote does not store these parameters in its non-volatile memory, therefore, if these parameters are used in operation they must be set at each mote boot.

Refer to the command descriptions in Chapter 2 for more information about the *setNVParameter* and *setParameter* commands.

Sending Data to and Receiving Data from the Network

For serial transmissions from the microprocessor to the network, users may choose between best-effort and reliable transport types.

Best-effort Communication

Packets requiring no explicit receipt acknowledgement may be sent through the network using a best-effort communication mechanism. This lowest overhead method is best suited when no application acknowledgement is required or when a small percentage of lost packets is tolerable. At the receiver, all packets with the best-effort delivery flag (see “Flags” on page 18) are forwarded to the application layer and no acknowledgements are generated. Consequently, the sender receives no feedback about the success of individual packet delivery. Applications requiring guaranteed delivery should use the reliable communication mechanism (see the following section, “Reliable Communication”).

Data received from the network is passed to the microprocessor by the *dataReceived* mote serial API command with the *flags* byte set to best effort, as described in “dataReceived” on page 60.

The microprocessor should always use best effort communication when it has a data packet to send through the network (as opposed to a response packet to a reliable request). For example, best-effort communication is recommended when periodically sending or bursting sensor data. The microprocessor sends a best-effort communication by using the *send* command with the *flags* byte set to best effort, as described in “send” on page 48.

Reliable Communication

In reliable communication, packets are acknowledged end-to-end, providing confirmation to the application layer that the packet was successfully delivered. If the confirmation is not received, the sender will retransmit the packet. Downstream communication refers to packets sent from the controller to the microprocessor; upstream communication refers to packets sent from the microprocessor to the controller (see Figure 3 on page 6). In the case of downstream reliable communication, the microprocessor must respond via the *send* command. A response from the microprocessor is required after receiving a *dataReceived* command with the *flags* byte set to *reliable* (see “dataReceived” on page 60). If the application layer has nothing to send to the mote, the payload should contain no data. Failure to send a response may result in the network manager taking corrective action, such as disconnecting the mote from the network.

Microprocessor-initiated reliable upstream is not supported.

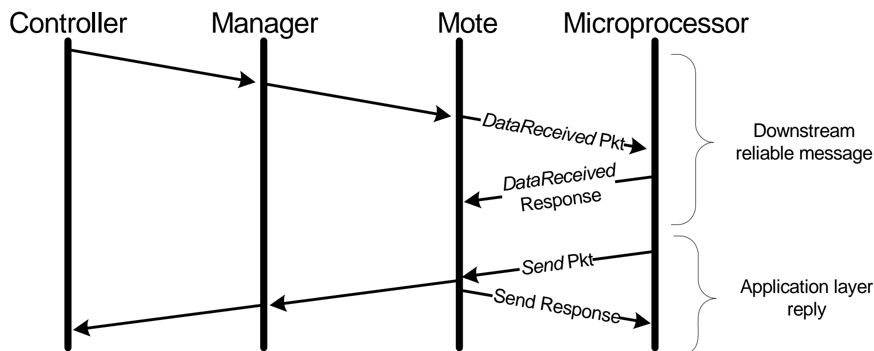


Figure 4 Reliable Communications from Controller to Microprocessor

Downstream Reliable Retries

Downstream reliable retries are handled in the following manner, as illustrated in Figure 5.

❶ When the mote receives a network packet, it forwards the payload, along with the transport sequence number via the *dataReceived* serial API command. The sensor processor can compare the sequence number with that of the previous packets to check if it is a duplicate.

❷ If the mote receives a duplicate packet, but has not received a *send* command in response, it forwards the message to the sensor processor.

❸ The sensor processor replies via the send serial API command and includes the sequence number and source address from the request along with the payload. The mote receives this and passes it on to the network.

❹ If the mote still has the buffered response from the sensor processor, it discards the duplicate and replies to the network on behalf of the sensor processor to close out the reliable acknowledgement.

❺ The mote will retain the buffered response until it receives a new data packet from the manager (with new sequence number), which signifies to the mote that the manager received the response to its previous packet.

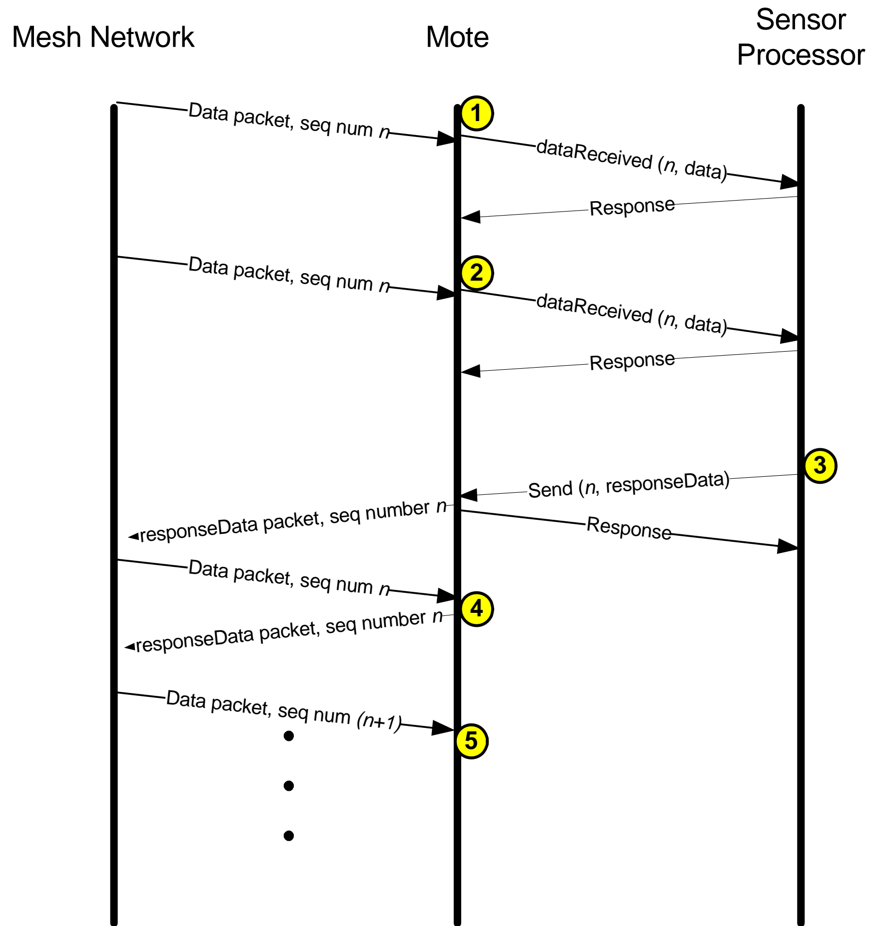


Figure 5 Downstream Reliable Retries

Application Support

Bandwidth Services

Service Types

IA-510 motes support bandwidth services to accommodate the dynamic bandwidth needs of the complex applications common in monitoring and control. The services mechanism provides a means by which the mote may request bandwidth from the network manager. When used in conjunction with a Dust Networks IA-510 network manager, the device receives the full benefits of bandwidth services to support the range of tasks that a fully productized device needs to perform-not just regular data reporting, but also rapid request-response configuration, block transfers, and alarm messages.

Services may be either of the following:

- Manager-originated-Bandwidth services pushed from the network manager to the device.
- Device-originated-Bandwidth services requested by the OEM microprocessor via the API commands, as described below.

Service-related API Commands

For device-originated services, the IA-510 mote makes the service requests based upon the values of the service table entries that are filled in by the microprocessor application via the services API. The IA-510 mote manages the service request packet exchange with the network manager (initial request, retries, and delayed response handling). The mote maintains a service table where each entry describes the service and the current service status (for example, whether the service is active). The mote serial API provides the following commands to manage services:

- **setParameter<service>**—This command enables a microprocessor to request a new service or modify an existing device-originated service. A successful response to this command indicates that the mote has updated its service table and will send a service request to the network manager once the mote is operational.
- **serviceIndication**—The mote sends this command to the microprocessor when it updates its service table after receiving notice of the following:
  - A new device-originated service was granted as a result of service request initiated by the microprocessor
  - A manager-originated service was created
  - An existing service was updated or deleted
- **getParameter<service>**—This command enables the microprocessor to check the services description and status of service table entries. The service table contains both manager-originated and device-originated services.

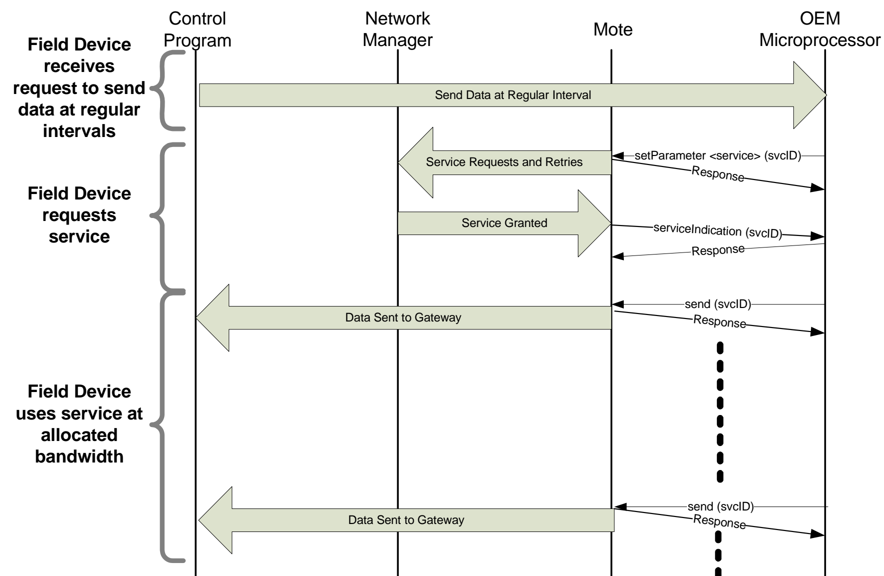


Figure 6 Transaction Diagram—Bandwidth Service Requests

The following are specific usage scenarios for requesting services.

- **Retrying service requests**—The response to a service request may take some time if it results in the network manager adding bandwidth to the network. The mote automatically handles communication retries with the network manager until it receives a response. If a service is denied, the microprocessor may re-send a service request to the network manager by reissuing the *setParameter<service>* request. A service may be denied if there is a problem in the network (for example, insufficient bandwidth upstream of the mote). The microprocessor may retry the service request after waiting five minutes to allow the network manager time to attempt to resolve the network problem.
- **Modifying an existing service**—If a service is active at rate X, and the microprocessor requests an update to a rate Y, the service state remains Active, but the service rate remains at rate X until the network manager responds to grant rate Y. The mote forwards a *serviceIndication* event to the microprocessor when it receives a notification from the network manager.
- **Deleting a service**—To delete a service, the microprocessor should issue a *setParameter<service>* command with a time value of zero. For more information, see “setParameter<service>” on page 28.
- **Network manager removes a service**—Dust network managers continuously optimize network links to maintain network performance in the face of RF challenges. However, in the event that a Dust network manager cannot sustain a service due to network conditions, it will reduce the allocated bandwidth to a level that effectively disables the service. The mote will receive notification from the network manager and forward a *serviceIndication* command to the microprocessor. The microprocessor may choose to submit another request to update the service at a future time. See “serviceIndication” on page 57.

Service Table

The mote service table describes both granted and requested services. The parameters below are used in the *setParameter<service>* and *getParameter<service>* commands.

- **Service ID**—A unique identifier for the service.
  - **Service IDs from 0x00 to 0x7F**—Indicate device-originated services that the microprocessor has requested.
  - **Service IDs from 0x80 to 0xFF**—Indicate manager-originated services.
- **Service State**—Used in *getParameter<service>*, this field reflects the state of the service. The fields are as follows:
  - **Status bit**—Indicates whether a service is inactive, active, or requested. Here, “requested” means OEM microprocessor has sent requested a service through *setParameter<service>*, but either the mote has not yet sent a request to the network manager, or the network manager has not accepted/ denied.
  - **Service pending bit**—Used for services in the requested state, the service-pending bit indicates if the mote has sent a service request to the network manager.

- **Service Flags**
  - **Source**—The mote is the source/sender of data.
  - **Sink**—The mote is the receiver of data.
  - **Intermittent/regular interval**—Designates whether the traffic will be used at regular intervals, or on an intermittent basis (for example, events).
- **Application Domain**
  - **Maintenance**—Symmetric data traffic, such as request-response pairs.
  - **Publish**—Data sent at a regular interval from mote to the gateway.
  - **Event**—Data sent infrequently from the mote to gateway (such as alarms).
  - **Block transfer**—Temporary high-speed transfer of large chunks of data.
- **Destination Address**—Because services are always between the gateway and the mote in SmartMesh IA-510, the destination address should be specified as 0xF981.
- **Time**—The time variable describes the rate of data.
  - If data is being regularly reported (the intermittent bit is not set on service request flag), then time is the period between data sent in milliseconds.
  - If the data is intermittent, then time is the desired latency in milliseconds.
  - To delete a service, set the *time* field of the desired service to zero. Service request flags, application domain, and destination address are ignored by the mote when time equals zero.

The following transaction diagram further illustrates how the service state byte is updated during both a successful and an unsuccessful service request.

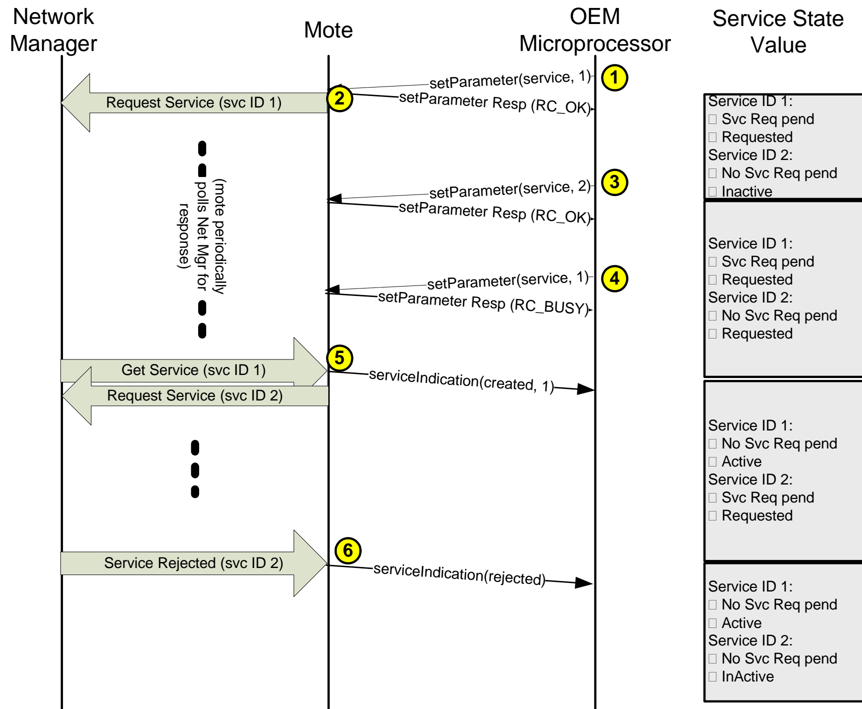


Figure 7 Transaction Diagram—Service State Values

❶ The OEM Microprocessor initiates a service request by calling *setParameter<service>* on an unused service ID.

❷ The mote receives the *setParameter<service>* command, updates its service table, and initiates a wireless request to the network manager. The mote handles all retries until it receives a success, warning, or error.

❸ The OEM microprocessor requests a service for service ID 2.

❹ The OEM microprocessor attempts to request a service ID 1. However, since there is a service request pending already for service ID 1, a serial API error is returned.

❺ The network manager grants the service. The mote updates its service table, and sends a *serviceIndication* packet to the OEM Microprocessor. The mote then initiates a request to the network manager for the next valid service in the service table.

❻ The network manager rejects the service. The mote updates its service table, and sends a *serviceIndication* packet to the OEM microprocessor.

Non-service Bandwidth Control

The mote may skip request services only if all of the following conditions are met:

- The dynamic bandwidth allocation is not required.
- The control application (communicating to the IA-510 network manager) uses the bandwidth control manager API commands to ensure bandwidth is always sufficient for the field device.

If the above conditions are met, then the OEM microprocessor is not required to request services, and may use service ID 0x80 when sending data via the *send* command (again, only if operating with a Dust Networks IA-510 network manager).

Timestamps

Many of the benefits of a SmartMesh IA-510 network, such as its high reliability and low power consumption, are attributable to the fact that it is a time synchronized mesh network. This means that under normal conditions every mote in the network has a shared sense of time accurate to better than a millisecond. Another advantage of time synchronization is that this sense of time is available to the microprocessor for application usage by means of the *getParameter<time>* command, which returns network time and Universal Time Clock (UTC) time.

Network time is specified relative to the Absolute Slot Number (ASN), which is the number of 10 ms timeslots since the manager booted. An ASN offset is also given, which is the number of microseconds since the beginning of the current slot. When using ASN and ASN offset, the accuracy of the timestamps network wide is better than a millisecond.

Each mote maintains a mote UTC reference, which is a mapping from ASN and ASN-offset into UTC time. The network periodically updates each mote's UTC reference and synchronizes it to the network real-time clock (NRTC), which serves as the timing master for the network and typically resides on the network manager. Note that there is some error introduced by this extra mapping from UTC to ASN due to the propagation delay of the UTC reference updates. The nominal UTC skew is within 100 ms. For this reason, network time (ASN) is recommended for applications requiring tight synchronization between motes, while UTC time is appropriate for time stamping relative to real-world events, such as data logging, where a high degree of accuracy is less important.

Events and Alarms

The mote serial API includes events and alarms that allow a microprocessor to have full visibility of mote states and conditions.

An alarm is an ongoing condition, such as low supply voltage or an error in non-volatile memory. Refer to the section “getParameter<moteStatus>” on page 33 for details on reading alarms and information about the available alarms.

By contrast, an event is defined as a discrete occurrence in mote or network operation. Examples of events include a mote startup, a mote failure to join the network, or a change in alarm condition, such as an alarm opening or closing.

Users can control which events are pushed to the microprocessor by using the *setParameter<eventMask>* command.

RF Compliance Testing Commands

The motes include commands useful for RF compliance testing or simple RF functional testing during OEM integration. See “testRadioTx” on page 53, “testRadioRx” on page 54, and “getParameter<testRadioRxStats>” on page 36. As Figure 8 shows, the *testRadioTx* and *testRadioRx* commands place the mote in radio test mode, and the mote returns to normal operation only after a hardware or software reset. Note that multiple radio test commands (*testRadioTx*, *testRadioRx*, and *testRadioRxStats*) can be issued while the mote is in radio test mode.

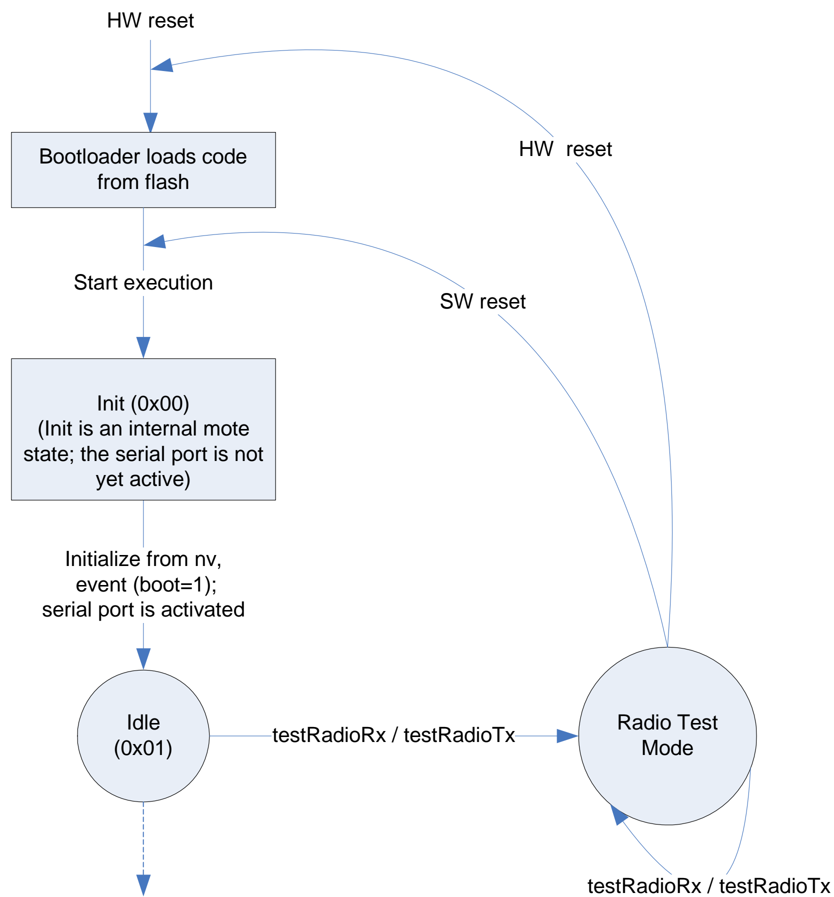


Figure 8 Mote State Diagram for RF Compliance Testing Commands

Mote Command Reference 2

This chapter describes each mote command and provides examples showing how HDLC packets are processed.

HDLC Packet Format

All mote HDLC commands are encapsulated in HDLC framing. Packets start and end with a 0x7E flag, and contain a frame checksum (FCS). Byte stuffing is used to escape occurrences of 0x7E values in the payload.

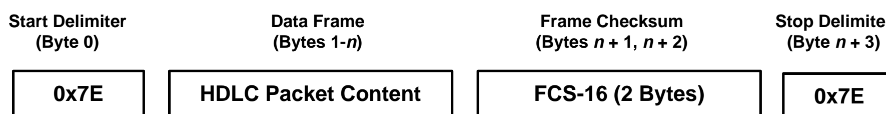


Figure 9 HDLC Packet Format

The FCS is calculated based on the 16-bit FCS computation method (FCS-16, RFC 1662). The mote checks the FCS and drops packets that have FCS errors. All numerical fields in a packet are in big endian order (MSB first), unless otherwise noted. An example of HDLC packet construction and HDLC packet decoding is provided in "HDLC Packet Processing Examples" on page 61.

Note: Some products may require an extra 0x7E start delimiter for proper operation at high bit rates. Refer to specific product datasheets for details.

Request and Response Packet Format

A command request consists of a command type and a structure containing the command arguments.

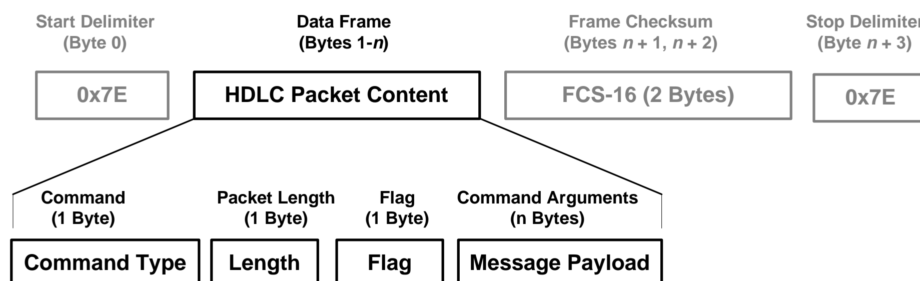


Figure 10 Command Request Format

A command response consists of a command type and a structure containing the response code and any data associated with the response.

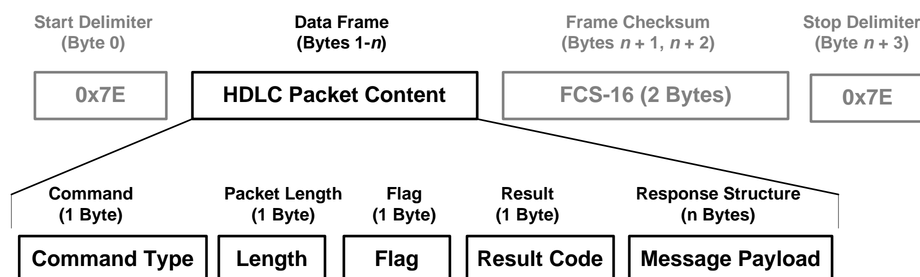


Figure 11 Command Response Packet

Command Type

The command type indicates which API message is contained in the message content. The message content for each command type is described later in this chapter.

Length

The length indicates the length of the message payload within the packet. The length of the message payload is command dependent, and does not include the bytes used for start delimiter, *command type*, *length* field, *flags* field, *result code* (if present), frame checksum, and stop delimiter.

Flags

The *flags* field contains the packet type (request or response), packet ID, and other command specific information.

Packet ID alternates for each new packet, and allows the mote or the microprocessor to distinguish between new and retransmitted packets it receives. If the ignore packet ID bit is set, the mote will ignore the packet ID and process each packet as new. For more information, see “Packet Retransmission” on page 2.

Table 1 Flag Bit Description

Bit	Description
7-3	Command-specific settings (see packet formats for each command)
2	Ignore packet ID: 0 = Do not ignore 1 = Ignore
1	Packet ID
0	Packet type: 0 = Request 1 = Response

Result Codes

Response packets may contain the following result codes.

Table 2 Result Codes

Value	Name	Description
0x00	RC_OK	Operation was successfully completed.
0x01	Reserved	Reserved.
0x02	Reserved	Reserved.
0x03	RC_BUSY	Operation on this service is in progress.
0x04	RC_INVALID_LEN	Invalid packet length.
0x05	RC_INV_STATE	Invalid mote state for command.
0x06	RC_UNSUPPORTED	Command not supported for hardware.
0x07	RC_UNKNOWN_PARAM	Unknown parameter value.
0x08	RC_UNKNOWN_CMD	Unknown command.
0x09	RC_WRITE_FAIL	Write did not complete.
0x0A	RC_READ_FAIL	Read did not complete.
0x0B	RC_LOW_VOLTAGE	Voltage check failed.
0x0C	RC_NO_RESOURCES	No resources are available to complete this command.
0x0D	RC_INCOMPLETE_JOIN_INFO	Incomplete join information.
0x0E	RC_NOT_FOUND	Parameter not found.
0x0F	RC_INVALID_VALUE	Invalid value.

Generic Acknowledgement Packet

All packets sent across the mote serial interface must be acknowledged. If the microprocessor or the mote receives a well-formed packet (with correct CRC) but with an unknown or unsupported command ID, it must reply with a result code of RC_UNKNOWN_CMD, as shown in the following generic acknowledgement packet.

Table 3 Generic Response for unknown command IDs

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	Echo back Cmd ID from request
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_UNKNOWN_CMD

If the mote sends a generic response, it will use the following bit values for the *Flags* byte:

Table 4 Flags Bit settings for Mote Generic Acknowledgement

Bit	Description
7-3	0
2	Ignore packet ID: 0 = Do not ignore
1	Packet ID (toggled)
0	Packet type: 1 = Response

Data Types

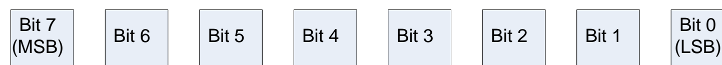
Packet formats described in this document use the following data types. The data type is shown in the *Data Type* column of the packet format description. *All data types are binary encoded*.

Table 5 Command Data Types

Data Type	Length
unsigned char (uint_8)	1 bytes
signed char (int_8)	1 byte
unsigned short (uint_16)	2 bytes
unsigned long (uint_32)	4 bytes
unsigned integer 40 (uint_40)	5 bytes
Byte Ordering	
Unsigned char (uint_8) and Signed char (int_8)	

Byte Ordering

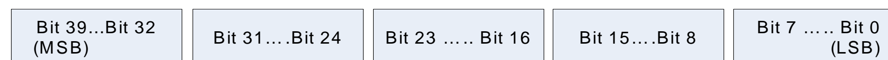
Unsigned char (uint_8) and Signed char (int_8)



Unsigned short (uint_16)

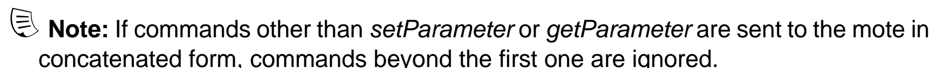
Unsigned long (uint_32)

Unsigned integer (uint_40)



Concatenating Command Payloads

The *setParameter* and *getParameter* commands allow concatenation of multiple payloads within a single request packet. This enables OEMs to optimize the number of command calls required to set or get information.



Note: If commands other than *setParameter* or *getParameter* are sent to the mote in concatenated form, commands beyond the first one are ignored.

The maximum number of parameters that may be requested is determined by both request and response packet size. Both the request and response packets have a maximum size of 128 bytes, including HDLC start and stop delimiters, frame checksum (as described above in HDLC Packet Format) as well as the command structure described here. If the response packet exceeds the limit, it is truncated to include only those responses that fit entirely. Note that byte-stuffing bytes, as illustrated in HDLC Packet Processing Examples, do not count against the limit.

For example, calling both *getParameter<time>* and *getParameter<charge>* commands would have the following concatenated request and response.

Table 6 getParameter<time & charge> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x0F
5	Cmd type	unsigned char	0x02
6	Length	unsigned char	0x01
7	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
8	Parameter type	unsigned char	0x10

Table 7 getParameter<time & charge> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x10
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK = OK RC_INVALID_LEN = Invalid packet length RC_UNKNOWN_PARAM = Unknown parameter
5	Parameter type	unsigned char	0x0F
6-9	UTC time seconds	unsigned long	
10-13	UTC time µseconds	unsigned long	
14-18	ASN	unsigned integer 40	
19-20	ASN offset	unsigned short	
21	Cmd type	unsigned char	0x02
22	Length	unsigned char	0x0B
23	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
24	Result	unsigned char	RC_OK = OK RC_INVALID_LEN = Invalid packet length RC_UNKNOWN_PARAM = Unknown parameter
25	Parameter type	unsigned char	0x10
26-29	Charge since last reset	unsigned long	Charge (mC)

Table 7 getParameter<time & charge> Response (Continued)

Msg Byte	Description	Data Type	Value
30-33	Uptime since last reset	unsigned long	Uptime (sec)
34	Temperature	signedChar	Temperature (°C)
35	Fractional temp	unsignedChar	Temperature in 1/255 of °C

Command Summary

Table 8 provides a summary of mote HDLC commands. For correct operation, all packets must be acknowledged. All unhandled packets should be acknowledged with an RC_UNKNOWN_CMD response code. The Destination column indicates whether the packet is sent (or received) through the network or processed locally by the mote. Refer to the product datasheet for details on serial handshake signals.

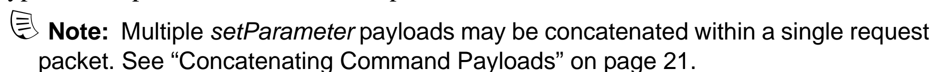
Table 8 Mote Command Summary

Type (HEX)	Command	Destination	Description
Microprocessor to Mote Commands			
0x01	setParameter	Local	Sets parameters on the mote.
0x02	getParameter	Local	Gets mote and network information.
0x03	setNVParameter	Local	Stores a given parameter in the mote's persistent storage.
0x04	getNVParameter	Local	Retrieves a given parameter from the mote's persistent storage.
0x05	send	Network	Packet destined for the network.
0x06	join	Local	Requests that mote attempt to join the network.
0x07	disconnect	Local	Requests that mote disconnect from the network.
0x08	reset	Local	Resets mote.
0x09	lowPowerSleep	Local	Shuts down peripherals and puts mote into deep sleep mode.
0x0B	testRadioTx	Local	Allows the serial device to initiate a series of packet transmissions.
0x0C	testRadioRx	Local	Allows the serial device to test radio reception for a specified channel.
0x10	clearNV	Local	Resets the mote non-volatile memory (NV) to its factory default state.
Mote to Microprocessor Commands			
0x0D	timeIndication	Local	Time and mote state information.
0x0F	events	Local	Notifies the microprocessor that a new event has occurred.
0x81	dataReceived	Network	Mote forwards a packet from the network to the microprocessor.

Microprocessor to Mote Commands

setParameter

The *setParameter* command can be used to set a number of configuration parameters in the mote. The *setParameter* response is sent within the $t_{diag_ack_timeout}$ (as defined in the product datasheet) of the request. The length of payload is dependant on the parameter type and is specified below for each parameter.



Note: Multiple *setParameter* payloads may be concatenated within a single request packet. See “Concatenating Command Payloads” on page 21.

Table 9 Set Parameter Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	Length of request (n+1)
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	Parameter type (see Table 11)
5	Data		First byte of data
...4+n	Data		Up to n-1 additional bytes of data

Table 10 Set Parameter Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	Length of response (n+1)
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK = Operation successfully completed RC_INVALID_LEN = Invalid packet length RC_UNKNOWN_PARAM = Unknown parameter
5	Parameter type	unsigned char	Parameter type (see Table 11)
6	Data		First byte of data
...5+n	Data		Up to n-1 additional bytes of data

Table 11 provides an overview of the parameters that may be used with the *setParameter* and *getParamete*r commands. These parameters are described in detail in the following sections.

Table 11 Parameters for setParameter and getParameter Commands

Value	Parameter Name	Description	Get	Set
0x04	txPower	Sets RF output power for the device.		√
0x06	joinDutyCycle	Allows the serial processor to control the ratio of active listen time to doze time (a low-power radio state) during network search.		√
0x07	batteryLife	Allows the serial processor to update the remaining battery life information that the mote reports to the manager.		√
0x08	service	Allows the device to initiate a service request or update an existing service.	√	√
0x0B	eventMask	Allows the serial process to disable events that may be sent in the mote's event notification message.		√
0x0C	moteInfo	Gets static information about the mote's hardware and software.	√	
0x0D	networkInfo	Gets the mote's current network-related parameters.	√	
0x0E	moteStatus	Gets the mote's state and frequently changing information.	√	
0x0F	time	Gets the current time on the mote.	√	
0x10	charge	Gets the mote's charge consumption.	√	
0x11	testRadioRxStats	Gets results of the mote radio reception test.	√	

setParameter<txPower>

The *setParameter<txPower>* command sets the mote conducted RF typical output power. Refer to product datasheets for supported RF output power values. For example, if the mote has a typical RF output power of +8 dBm when the power amplifier (PA) is enabled, set the *txPower* parameter to 8 to enable the PA. Similarly, if the mote has a typical RF output power of -2 dBm when the PA is disabled, then set the *txPower* parameter to -2 to turn off the PA. Note that this value is the RF output power coming out of the mote and not the radiated power coming out of the antenna. This command may be issued at any time and takes effect upon the next transmission.

Table 12 setParameter<txPower> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	0x02
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x04
5	Transmit power	signed char	

Table 13 setParameter<txPower> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNKNOWN_PARAM RC_INVALID_VALUE
5	Parameter type	unsigned char	0x04

setParameter<joinDutyCycle>

The *setParameter<joinDutyCycle>* command allows the microprocessor to control the join duty cycle-the ratio of active listen time to doze time (a low-power radio state) during the period when the mote is searching for the network. By default, the mote uses a 5% join duty cycle, which enables the mote to join the network at a reasonable rate without using excessive battery power. If you desire a faster join time at the risk of higher power consumption, use the *setParameter<joinDutyCycle>* command to increase the join duty cycle up to 100%. Note that the *setParameter<joinDutyCycle>* command is not persistent and affects only the next join. For power consumption information, refer to the mote product datasheet.

This command may be issued multiple times during the joining process. This command is only effective when the mote is in the IDLE and SEARCHING states.

Table 14 setParameter<joinDutyCycle> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	0x02
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x06
5	Join duty cycle	unsigned char	0 to 255, where 0 = 0% 255 = 100%

Table 15 setParameter<joinDutyCycle> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNKNOWN_PARAM
5	Parameter type	unsigned char	0x06

setParameter<batteryLife>

The *setParameter<batteryLife>* command allows the microprocessor to update the remaining battery life information that the mote reports to the network manager. This parameter must be set during the IDLE state prior to joining, and should be updated periodically throughout operation.

Table 16 setParameter<batteryLife> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	0x04
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x07
5-6	Estimated remaining battery life (days)	unsigned short	
7	Power status	unsigned char	0x00 = Nominal 0x01 = Low 0x02 = Critically low 0x03 = Recharging low 0x04 = Recharging high

Table 17 setParameter<batteryLife> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNKNOWN_PARAM
5	Parameter type	unsigned char	0x07

setParameter<service>

The *setParameter<service>* command is used to request new device-originated bandwidth services and modify existing device-initiated services. For more information about services, see “Bandwidth Services” on page 9.

Calling the *setParameter<service>* command updates the mote’s internal service table, which later initiates a request via the network to the network manager for a bandwidth service. The response packet signifies receipt of the request by the mote. A subsequent serviceIndication notification will be sent indicating the response from the network manager. The *getParameter<service>* command may be used to read the service table, including the state of the service request.

The *setParameter<service>* command may be sent at any time. If the network manager rejects a service request, the microprocessor can try again by re-issuing the setParameter<service> command.

For details on each of the parameters, see “Service Table” on page 11.

For supported combinations of the service request flags by application domain, see Table 21.

To delete a service, set the *time* field of the desired service to zero. Service request flags, application domain, and destination address are ignored by the mote when time equals zero.

Note: For restrictions on when the *setParameter<service>* command may be called, see “Bandwidth Services” on page 9.

Table 18 setParameter<service> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	0x0A
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x08
5	Service ID	unsigned char	0x00-0x7F
6	Service request flags	unsigned char	See Table 20
7	Application domain	unsigned char	Service application domains. 0x00 = Publish 0x01 = Event 0x03 = Block transfer
8-9	Destination address	unsigned short	0xF981 = gateway
10-13	Time	unsigned long	Period (msec), or latency (msec) if intermittent flag is set. See “Service Table” on page 11 for details.

Table 19 setParameter<service> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	0x02
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_BUSY RC_INVALID_LEN RC_UNKNOWN_PARAM RC_INVALID_VALUE RC_NO_RESOURCES = service table is full
5	Parameter type	unsigned char	0x08
6	Number of remaining service entries on mote	unsigned char	

Table 20 Service Request Flags Bitmask Values

BitMask	Name	Description
0x01	Source	Mote is source of data generated
0x02	Sink	Mote is receiver of data
0x04	Intermittent	Intermittent traffic (as opposed to regular reporting)

Table 21 Valid Bit Values of Service Parameters by Application Domain

Application Domain	Source	Sink	Intermittent
Publish	1	0	0
Event	1	0	1
Block Transfer	0 or 1	0 or 1	0
Maintenance*			
* Any value for maintenance application domain will be denied.			

setParameter<eventMask>

The *setParameter<eventMask>* command allows the microprocessor to mask events that may be sent in the mote's events notification message. This command may be called at any time and takes effect at the next event packet. The mote includes an event in the notification message if the corresponding bit in <eventMask> is set to "1," and excludes the event if the bit is set to "0." At mote reset, the default value of <eventMask> is "1" for all events.

Table 22 setParameter<eventMask> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	0x05
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x0B
5-8	Event mask	unsigned long	

Table 23 setParameter<eventMask> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x01
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_UNKNOWN_PARAM
5	Parameter type	unsigned char	0x0B

Table 24 Event Bitmask Values

BitMask	Event	Description
0x01	Boot	The mote booted up.
0x02	Alarms changed	Value of alarms field changed.
0x08	Join failed	Join operation failed.
0x10	Disconnected	The mote has disconnected from the network.
0x20	Operational	The mote has a connection to the gateway to send data.

getParameter

The *getParameter* command allows a number of configuration parameters in the mote to be read by serial. When a *getParameter* command is sent, the response to the request is sent within the $t_{diag_ack_timeout}$ (as defined in the product datasheet). The command structure for individual parameter types can be found below.

Table 11 provides an overview of the parameters that may be used with the *getParameter* command. These parameters are described in detail in the following sections.

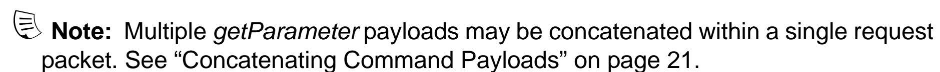

Note: Multiple *getParameter* payloads may be concatenated within a single request packet. See “Concatenating Command Payloads” on page 21.

Table 25 getParameter Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	Length of request (n+1)
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	Parameter type (see Table 11)
5	Data		First byte of data
...4+n	Data		Up to n-1 additional bytes of data

Table 26 getParameter Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	Length of response (n+1)
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNKNOWN_PARAM
5	Parameter type	unsigned char	Parameter type (see Table 11)
6	Data		First byte of data
...5+n	Data		Up to n-1 additional bytes of data

getParameter<moteInfo>

The *getParameter<moteInfo>* command returns static information about the mote's hardware and software. Note that network state-related information about the mote may be retrieved using *getParameter<networkInfo>. As with all commands, all field data is binary encoded*.

Table 27 getParameter<moteInfo> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x0C

Table 28 getParameter<moteInfo> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x11
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNKNOWN_PARAM
5	Parameter type	unsigned char	0x0C
6	API version	unsigned char	
7-14	Serial number	Array of 8 unsigned char	
15	HW model	unsigned char	Hardware model
16	HW revision	unsigned char	Hardware revision
17	SW major revision	unsigned char	Major software revision
18	SW minor revision	unsigned char	Minor software revision
19	SW patch	unsigned char	Software patch number
20-21	SW build	unsigned short	Software build number

getParameter<networkInfo>

The *getParameter<networkInfo>* command may be used to retrieve the mote's network-related parameters. Note that static information about the mote's hardware and software may be retrieved using *getParameter<moteInfo>*.

MAC address—The MAC address is the vendor IEEE address used by the mote. This value defaults to the factory-configured serial number, unless overwritten via *setNVParameter<macAddress>* command.

Mote ID—An ordinal number (1, 2, 3 …), the *mote ID* is a short address assigned to the mote by the manager at network join time.

Table 29 getParameter<networkInfo> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x0D

Table 30 getParameter<networkInfo> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x0D
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNKNOWN_PARAM
5	Parameter type	unsigned char	0x0D
6-13	MAC Address	Array of 8 unsigned char	
14-15	Mote ID	unsigned short	
16-17	Network ID	unsigned short	

getParameter<moteStatus>

The *getParameter<moteStatus>* command is used to retrieve the mote state and frequently changing information. Note that static information about the state of the mote hardware and software may be retrieved using *getParameter<moteInfo>*.

Table 31 getParameter<moteStatus> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x0E

Table 32 getParameter<moteStatus > Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x0B
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNKNOWN_PARAM
5	Parameter type	unsigned char	0x0E
6	Mote state	unsigned char	See Table 33
7	Mote state reason	unsigned char	Reserved

Table 32 getParameter<moteStatus > Response (Continued)

Msg Byte	Description	Data Type	Value
8-9	Change counter	unsigned short	
10	Number of parents	unsigned char	
11-14	Current mote alarms	unsigned long	See Table 34
15	Reserved	unsigned char	

Table 33 Mote States

State #	Description	Details
0x00	Init	The mote is in the process of booting.
0x01	Idle	The mote is accepting configuration commands. Upon receiving a join command, the mote moves into the Searching state.
0x02	Searching	The mote's receiver is on with a configurable duty cycle while the mote is actively searching for a network.
0x03	Negotiating	The mote has detected a network and has received a join request from the manager.
0x04	Connected	The mote has joined the network and established communication with the network manager.
0x05	Operational	The mote has links to both the network manager and gateway, and is ready to send data.
0x06	Disconnected	The mote has disconnected from the network.

Table 34 Alarm Bitmask Values

BitMask	Event	Description
0x01	NV Error	Non-volatile error.
0x02	Low voltage	Mote supply voltage is below operating minimum.
0x04	OTP Error	Detected an error in factory-programmed settings.

getParameter<time>

The *getParameter<time>* command is used to request the current time on the mote.

UTC time—UTC time is composed of two components: UTC time seconds and UTC time microseconds. UTC time seconds is the number of seconds since midnight of January 1, 1970. UTC time microseconds is the microseconds since the beginning of the current second. UTC time is propagated from the manager, and therefore may depend on an accurate external time source, such as NTP.

ASN—ASN is the absolute slot number-the number of timeslots since the access point last reset.

ASN offset—The *ASN offset* is the number of microseconds since the beginning of the current slot.

Table 35 getParameter<time> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x0F

Table 36 getParameter<time> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x10
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNKNOWN_PARAM
5	Parameter type	unsigned char	0x0F
6-9	UTC time seconds	unsigned long	
10-13	UTC time microseconds	unsigned long	
14-18	ASN	unsigned integer 40	
19-20	ASN offset	unsigned short	

getParameter<charge>

The *getParameter<charge>* command retrieves the charge consumption of the mote since the last reset, and the mote uptime and last measured temperature.

Table 37 getParameter<charge> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x10

Table 38 getParameter<charge> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x0B
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNKNOWN_PARAM
5	Parameter type	unsigned char	0x10
6-9	Charge since last reset	unsigned long	Charge (mC)
10-13	Uptime since last reset	unsigned long	Uptime (sec)
14	Temperature	signedChar	Temperature (°C)
15	Fractional temp	unsignedChar	Temperature in 1/255 of °C

getParameter<testRadioRxStats>

The *getParameter<testRadioRxStats>* command retrieves statistics for the latest radio reception test performed using the *testRadioRx* command. The statistics show the number of good and bad packets (CRC failures) received during the test.

Table 39 getParameter<testRadioRxStats> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x11

Table 40 getParameter<testRadioRxStats> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x05
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNKNOWN_PARAM
5	Parameter type	unsigned char	0x11
6-7	Number of packets received okay	unsigned short	
8-9	Number of reception failures	unsigned short	

getParameter<service>

The *getParameter<service>* command retrieves information about the service allocation that is currently available to the field device.

Note that since the mote sends the microprocessor a *serviceIndication* whenever the manager adds or updates a service, reading the service table via *getParameter <service>* should not be necessary.

For details on each of the parameters, see “Service Table” on page 11.

Table 41 getParameter<service> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x02
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Parameter type	unsigned char	0x08
5	Service ID	unsigned char	0x00-0xFF

Table 42 getParameter<service> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x02
2	Length	unsigned char	0x0B
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNKNOWN_PARAM RC_NOT_FOUND = service not found
5	Parameter type	unsigned char	0x08
6	Service ID	unsigned char	0x00-0xFF
7	Service state	unsigned char	Service pending bit (bit 7): 0 = No service pending 1 = Service request pending Status bit (bits 6-0): 0x00 = Inactive 0x01 = Active 0x02 = Requested
8	Service flags	unsigned char	See Table 20
9	Application domain	unsigned char	0x00 = Publish 0x01 = Event 0x02 = Maintenance 0x03 = Block transfer
10-11	Destination address	unsigned short	
12-15	Time	unsigned long	

setNVParameter

The *setNVParameter* command stores a given parameter in the mote's persistent storage. The values stored in NV will be used starting with the next mote boot. Some parameters may be safely updated without a mote reboot, so the *setNVParameter* command has an option to also update the value of the parameter in mote RAM. For details on the effects of updating RAM, see the descriptions of the individual commands below.

Writing non-volatile memory causes a temporary increase in current consumption and the flash on the mote has a limited number of lifetime writes. For these reasons, the *setNVParameter* command should be used sparingly.

Table 43 setNVParameter Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	Length of request
3	Flags	unsigned char	See Table 45 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	See Table 46
9…	Parameter value		

Table 44 setNVParameter Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_WRITE_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	See Table 46

Table 45 setNVParameter Request Flag Bit Description

Bit	Description
7	Write to RAM: 0 = Write NV only 1 = Write NV and RAM
6-3	Reserved (use 0's for these bits)
2-0	See Table 1 for bit description

Table 46 Parameters for setNVParameter and getNVParameter Commands

Value	Parameter Name	Description	GetNV	SetNV
0x01	macAddress	Sets the MAC address in the non-volatile memory of the mote.	√	√
0x03	networkId	Sets the mote's network ID in the non-volatile memory of the mote.	√	√
0x04	txPower	Sets RF output power for the device in the non-volatile memory of the mote.	√	√
0x02	joinKey	Sets the network join key in the non-volatile memory of the mote. (Not readable for security reasons.)		√
0x05	powerInfo	Sets the power supply for the device.	√	√
0x15	OTAPlockout	Sets whether a mote can be over-the-air programmed (OTAP).	√	√

setNVParameter<macAddress>

The *setNVParameter<macAddress>* command may be used to configure the *MAC address* of the mote in non-volatile memory. The RAM bit should be set if calling this command prior to join, when the mote is in the IDLE state.

Table 47 setNVParameter<macAddress> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x0D
3	Flags	unsigned char	See Table 45 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	0x01
9-16	MAC address	Array of 8 unsigned char	

Table 48 setNVParameter<macAddress> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_WRITE_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	0x01

setNVParameter<networkID>

The network ID is the identification number used to distinguish different wireless networks. In order to join a specific network, the mote must have the same network ID as the network manager. Upon receiving this request, the mote stores the new network ID in its persistent storage area for use at the next mote boot. The RAM value should be used prior to join, when the mote is in the IDLE state. However, occasionally it may be necessary to attempt to change the network ID after the manager has issued the *join* command. For more information, see Appendix A.

Table 49 setNVParameter<networkID > Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x07
3	Flags	unsigned char	See Table 45 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	0x03
9-10	Network ID	unsigned short	

Table 50 setNVParameter<networkID Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_WRITE_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	0x03

setNVParameter<txPower >

The *setParameter<txPower>* command sets the mote output power. Refer to product datasheets for supported RF output power values. For example, if the mote has a typical RF output power of +8 dBm when the power amplifier (PA) is enabled, then set the *txPower* parameter to 8 to enable the PA. Similarly, if the mote has a typical RF output power of -2 dBm when the PA is disabled, then set the *txPower* parameter to -2 to turn off the PA. This command may be issued at any time and takes effect at the next mote boot. To change the transmit power immediately, use the *write* RAM option of this command, which can also be used at any time.

Table 51 setNVParameter<txPower> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x06
3	Flags	unsigned char	See Table 45 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	0x04
9	Transmit power	signed char	

Table 52 setNVParameter<txPower> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_WRITE_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	0x04

setNVParameter<joinKey >

The *setParameter<joinKey>* command may be used to set the join key in mote persistent storage that the mote uses to establish a secure network connection. The join key is required for a mote to join a network and is specific for the network. Upon receiving this request, the mote stores the new join key in its persistent storage. Using the *write* RAM option will only have an effect if the command is called while the mote is in IDLE mode. Otherwise, the new value will be used after the next mote boot.

Table 53 setNVParameter<joinKey > Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x15
3	Flags	unsigned char	See Table 45 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	0x02
9-24	New join key	Array of 16 unsigned char	New join key

Table 54 setNVParameter<joinKey > Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_WRITE_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	0x02

setNVParameter<powerInfo>

The *setNVParameter<powerInfo>* command specifies the amount of the power supply that is available to the mote. Using the *write* RAM option will only have an effect if the command is called while the mote is in IDLE mode. Otherwise, the new value will be used after the next mote boot.

Power source—The *power source* field describes whether the device is line, battery, or rechargeable/scavenging powered. A line-powered device has a hardwired connection to a plant power source. A battery-powered device operates solely off its (internal) battery. A device powered by rechargeable battery power or by energy scavenging has a short-term supply of energy (at least one hour) that is continuously being replenished. The device's power is being replenished by harvesting and converting energy from the environment surrounding the device (for example, solar, vibration, or heat).

Discharge current—The *discharge current* field indicates the maximum average current available to the mote. The manager will assign links such that this maximum average will not be exceeded, but may assign much lower as required by topology and data rates. This may limit the function of the device. For example, high-speed pipes may not be available if their use would exceed the discharge current.

Discharge time—The *discharge time* field specifies the duration that the power source can supply the discharge current. In the case where the device is line-powered, or has a battery that will last longer than $2^{32} * 1/32$ ms (approximately 37 hours), the discharge time will not be used in link assignment and should be set to 0xFFFF FFFF.

Recover time—The *recover time field* specifies the time the device power supply takes to recharge under no load before the discharge current is available. If the *power source* is not rechargeable, recover time should be set to zero.

Table 55 setNVParameter<powerInfo> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x10
3	Flags	unsigned char	See Table 45 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	0x05
9	Power source	unsigned char	0x00 = Line 0x01 = Battery 0x02 = Rechargeable/ Scavenging
10-11	Discharge current (µA)	unsigned short	
12-15	Discharge time (sec)	unsigned long	
16-19	Recover time (sec)	unsigned long	

Table 56 setNVParameter<powerInfo> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_WRITE_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	0x05

setNVParameter<OTAPlockout>

The *setNVParameter<OTAPlockout>* command specifies whether the mote can be over-the-air programmed (OTAP). When *OTAP lockout* is set to 0x00, OTAP is allowed; when it set to 0x01, OTAP is disabled (by default, *OTAP lockout* is set to 0x00). If the Write to RAM bit of the *flags* field (see Table 45) is set, the value in RAM is updated and will take effect on the next OTAP. Note that if the setting is changed while an OTAP is in progress, the new setting will not affect the current OTAP. If the Write to RAM bit is not set, the new setting takes effect after the mote reboots.

Dust Networks recommends that OEMs allow their devices to receive OTAPs by leaving the *OTAPlockout* parameter at its default value (OTAP allowed).

Table 57 setNVParameter<OTAPlockout> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x06
3	Flags	unsigned char	See Table 45 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	0x15
9	OTAP lockout	unsigned char	0x00 = OTAP allowed (default) 0x01 = OTAP disabled

Table 58 setNVParameter<OTAPlockout> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x03
2	Length	unsigned char	0x01
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_WRITE_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	0x15

getNVParameter

The *getNVParameter* command reads a parameter from the mote's persistent storage. For parameters that have never been written to the mote's persistent storage, a READ_FAIL result code will be returned. Note that if a value is updated using the *setNVParameter* command and the *flag* bit is not set to write the value to RAM, the value returned using the *getNVParameter* command will be different from the RAM value (the value currently in use).

Table 59 getNVParameter Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	0x05
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	See Table 46

Table 60 getNVParameter Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	Length of response
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_NOT_FOUND RC_READ_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	See Table 46
6…	Value	Array of unsigned char	

getNVParameter<macAddress>

The *getNVParameter<macAddress>* command returns the MAC address stored in mote non-volatile memory.

Table 61 getNVParameter<macAddress> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	0x05
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	0x01

Table 62 getNVParameter<macAddress> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	0x09
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_NOT_FOUND RC_READ_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	0x01
6-13	MAC address	Array of 8 unsigned char	

getNVParameter<networkID>

The *getNVParameter<networkID>* command returns the network ID stored in mote non-volatile memory.

Table 63 getNVParameter<networkID> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	0x05
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	0x03

Table 64 getNVParameter<networkID > Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	0x03
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_NOT_FOUND RC_READ_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	0x03
6-7	Network ID	unsigned short	

getNVParameter<txPower >

The *getNVParameter<txPower>* command returns the transmit power stored in mote non-volatile memory.

Table 65 getNVParameter<txPower> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	0x05
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	0x04

Table 66 getNVParameter<txPower> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	0x02
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_NOT_FOUND RC_READ_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	0x04
6	Transmit power	signed char	

getNVParameter<powerInfo>

The *getNVParameter<powerInfo>* command describes the power supply that is available to the mote.

Table 67 getNVParameter<powerInfo> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	0x05
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	0x05

Table 68 getNVParameter<powerInfo> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	0x0C
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_NOT_FOUND RC_READ_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	0x05
6	Power source	unsigned char	0x00 = Line 0x01 = Battery 0x02 = Rechargeable/ Scavenging
7-8	Discharge current (µA)	unsigned short	
9-12	Discharge time (sec)	unsigned long	
13-16	Recharge time (sec)	unsigned long	

getNVParameter<OTAPlockout>

The *getNVParameter<OTAPlockout>* command reads the OTAP lockout setting from the mote's persistent storage. OTAP lockout specifies whether the mote can be over-the-air programmed (OTAP).

Table 69 getNVParameter<OTAPlockout> Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	0x05
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4-7	Reserved	unsigned long	0x0000 0000
8	NVParameter ID	unsigned char	0x15

Table 70 getNVParameter<OTAPlockout> Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x04
2	Length	unsigned char	0x02
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_NOT_FOUND RC_READ_FAIL RC_LOW_VOLTAGE
5	NVParameter ID	unsigned char	0x15
6	OTAP lockout	unsigned char	0x00 = OTAP allowed (default) 0x01 = OTAP disabled

send

The *send* command allows the serial device to send a packet into the network through the mote serial port. The mote forwards the packet to the network upon receiving it. The microprocessor must not attempt to send data at a rate that exceeds its allocated bandwidth.

The *send* command is only valid when the mote is in the OPERATIONAL state. If the mote receives this command when it is not in the OPERATIONAL state, it returns the error RC_INV_STATE. It is possible that the serial device could receive a request while the mote is in the process of transition from the CONNECTED state to the OPERATIONAL state.

Flags—To use reliable transport, the sending device must set the transport type bit and the transport direction bit in the header (see Table 73) to distinguish between an end-to-end request and a response. The sender may specify the type of packet using the application domain parameter.

Destination address—If the *send* command is a response to a *dataReceived* command request, the destination address should be set to the source address from the *dataReceived* command. Otherwise, use 0xF981 for the destination address.

Service ID—The service ID must specify a valid service.

Application domain—For an end-to-end response to a reliable *dataReceived* command request, the application domain should be set to “maintenance.”

Priority—The mote maintains a queue of packets to be sent into the network. The queue is sorted in order of priority.

Sequence number—The sequence number is required when sending an end-to-end reliable response packet. This number must be the sequence number of the end-to-end request packet previously received via the *dataReceived* command. When sending a reliable request, the mote manages the sequence number on behalf of the network device. Therefore, if sending a reliable request or an unreliable packet use the value 0xFF for the sequence number.

Payload length—The maximum length of the message payload is 94 bytes.

Payload—The format of the serial packet payload is transparent to the mote.

Table 71 send Request

Msg Byte	Description	Data Type	Request (Sent to Mote)
1	Cmd type	unsigned char	0x05
2	Length	unsigned char	Length of request
3	Flags	unsigned char	See Table 73 Bit 0 = 0 (request)
4-5	Destination address	unsigned short	
6	Service ID	unsigned char	
7	Application domain	unsigned char	0x00 = Publish 0x01 = Event 0x02 = Maintenance 0x03 = Block transfer
8	Priority	unsigned char	0x00 = Low 0x01 = Medium 0x02 = High
9-10	Reserved	unsigned short	0
11	Sequence number	unsigned char	
12	Payload length	unsigned char	
13…	Payload	Array of unsigned char	

Table 72 send Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x05
2	Length	unsigned char	0x00
3	Flags	unsigned char	0x01
4	Result	unsigned char	RC_OK RC_INVALID_LEN RC_UNK_PARAM RC_INVALID_VALUE = Invalid application domain or priority RC_NOT_FOUND = Service or route destination not found RC_NO_RESOURCES = Mote buffers are full RC_INV_STATE = Mote is not in OPERATIONAL state

Table 73 send Request-Flag Bit Description

Bit	Description
7	Transport direction: 0 = End-to-end request 1 = End-to-end response
6	Transport type: 0 = Best effort 1 = Reliable *
5-3	Reserved (use 0's for these bits)
2-0	See Table 1
*Only used for end-to-end response to a reliable dataReceived command request (see Figure 4). Microprocessor-initiated end-to-end reliable requests are not supported.	

join

The *join* command requests that a mote start searching for the network and attempt to join. The mote must be in the IDLE state for this command to be valid. The join time is partly determined by the join duty cycle. For guidance on setting this parameter, see “setParameter<joinDutyCycle>” on page 26.

Table 74 join Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x06
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)

Table 75 join Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x06
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK = OK RC_INV_STATE = Invalid state for join RC_INCOMPLETE_JOIN_INFO = Incomplete join information

disconnect

The *disconnect* command requests that the mote disconnect from the network. The mote will send an indication to its network neighbors that it is about to become unavailable. Just after the mote disconnects, it sends the microprocessor an events packet with the disconnected bit set, indicating it will reset.

This command is only valid in when the mote is in the Connected or Operational state.

Table 76 disconnect Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x07
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)

Table 77 disconnect Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x07
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK

reset

Upon receiving this command, the mote resets itself. The delay to the actual reset will be two seconds. The mote will always send a response packet before initiating the reset. To force the mote to gracefully leave the network, use the *disconnect* command (see “disconnect” on page 51).

Table 78 reset Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x08
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)

Table 79 reset Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x08
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK

lowPowerSleep

The *lowPowerSleep* command shuts down all peripherals and places the mote in deep sleep mode. The command executes after the mote sends its response. The mote enters deep sleep within two seconds after the command executes. The command may be issued at any time and will cause the mote to interrupt all in-progress network operation. To achieve a graceful disconnect, use the disconnect command before using the *lowPowerSleep* command. A hardware reset is required to bring a mote out of deep sleep mode (see Figure 2). For power consumption information, refer to the mote product datasheet.

Table 80 lowPowerSleep Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x09
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)

Table 81 lowPowerSleep Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x09
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK

testRadioTx

The *testRadioTx* command allows the microprocessor to initiate a series of packet transmissions. Each packet transmission is 128 bytes in length. Byte 0 contains the packet length (excluding the length parameter itself), which is 127 bytes. Bytes 1 and 2 contain the sequence number that increment with every packet transmitted. Bytes 3-125 contain a big-endian counter (from 0-122) that increments with every byte. The interpacket delay is 20 ms. The *testRadioTx* command may only be issued when the mote is in IDLE mode, prior to its joining the network. The mote must be reset (either hardware or software reset) after radio tests are complete and prior to joining. Because a hardware reset causes the mote to load code from flash to RAM, its execution time is typically longer than for a software reset. Refer to the mote datasheet for hardware reboot time. Figure 8 provides a mote state diagram for the *testRadioTx* command.

If number of packets parameter is set to 0x00, the mote will generate an unmodulated test tone on the selected channel. The test tone can only be stopped by resetting the mote.

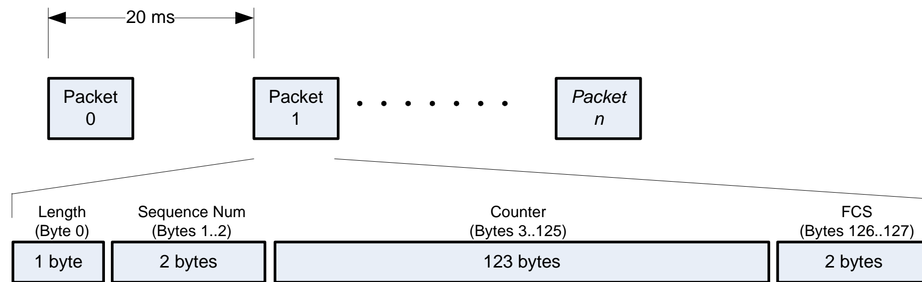


Figure 12 testRadioTx Packet Format

Table 82 testRadioTx Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x0B
2	Length	unsigned char	0x03
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Channel to use for transmission	unsigned char	RF channel 0-15
5-6	Number of packets to send	unsigned short	0x00 = Unmodulated test tone

Table 83 testRadioTx Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x0C
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_UNKNOWN_CMD RC_NO_RESOURCES = Mote buffers are full RC_INV_STATE = The mote is not in IDLE state RC_BUSY = Another test operation is in progress

testRadioRx

The *testRadioRx* command clears all previously collected statistics and initiates a test of radio reception for the specified channel and duration. During the test, the mote keeps statistics of the number of packets received (with and without error). The test results may be retrieved using the *getParameter<testRadioRxStats>* command. This command may only be issued in IDLE mode. The mote must be reset (either hardware or software reset) after radio tests are complete and prior to joining. Because a hardware reset causes the mote to re-initialize all hardware, its execution time is typically longer than for a software reset. Refer to the mote datasheet for hardware reboot time. Figure 8 provides a mote state diagram for the *testRadioRx* command.

Table 84 testRadioRx Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x0C
2	Length	unsigned char	0x03
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Channel to be tested	unsigned char	RF channel 0-15
5-6	Duration of test	unsigned short	Test duration in seconds

Table 85 testRadioRx Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x0C
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_INVALID_LEN = Invalid length of request packet RC_INVALID_VALUE = A parameter value is not valid RC_BUSY = Another test operation is in progress RC_INV_STATE = The mote is not in IDLE state

clearNV

The *clearNV* command resets the mote's non-volatile memory (NV) to its factory-default state (see Table 88). The mote NV contains the minimum set of parameters required for joining. For more information, see "Configuring the Mote" on page 6.

Table 86 clearNV Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x10
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)

Table 87 clearNV Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x10
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK = OK RC_INVALID_LEN RC_WRITE_FAIL = Flash operation failed RC_LOW_VOLTAGE = Low voltage

Table 88 Mote NV Factory Default Values

Parameter	Factory Default Value
macAddress	8 byte UID value from OTP
txPower	+8 dBm
networkId	1229
joinKey	0x445553544E4554574F524B53524F434B
nonceCounter	0
joinDutyCycle	5%
powerInfo.powerSrc	battery
powerInfo.dischargeCurrent	1000 µA (1 mA)
powerInfo.dischargeTime	0x6DDD00 (2000 hours, functionally equivalent to 0xFFFFFF)
powerInfo.recoverTime	0

Mote to Microprocessor Commands

timeIndication

This command applies to mote products that support a time interrupt into the mote. Refer to the mote datasheet for product-specific details. The time packet includes the network time and the current real time relative to the manager. The mote sends this response when the $\overline{\text{TIME}}$ pin is strobed high to low for a minimum of min_strobe_length, as defined in the product datasheet. The datasheet also provides information about $\overline{\text{TIME}}$ pin usage.

UTC time—UTC time is composed of two components: UTC time seconds and UTC time microseconds. UTC time seconds is the number of seconds since midnight of January 1, 1970. UTC time microseconds is the microseconds since the beginning of the current second.

ASN—ASN is the absolute slot number, the number of timeslots since the manager booted.

ASN offset—The ASN offset is the number of microseconds since the beginning of the current slot.

Table 89 timeIndication Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x0D
2	Length	unsigned char	0x0F
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4-7	UTC time seconds	unsigned long	
8-11	UTC time microseconds	unsigned long	
12-16	ASN	unsigned integer 40	
17-18	ASN offset	unsigned short	

Table 90 timeIndication Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x0D
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_NO_RESOURCES

serviceIndication

The *serviceIndication* command notifies the microprocessor of new manager-originated services, or changes in existing services. For more information on when the *serviceIndication* command is sent and details about the individual parameters, see “Bandwidth Services” on page 9.

If the *time* field contains the value 0x07FF FFFF, it indicates that the manager is unable to sustain the service due to network conditions and has effectively disabled the service. The service is not removed, however, and the microprocessor can elect to either delete the service or submit a request to update the service at a future time.

Table 91 serviceIndication Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x0E
2	Length	unsigned char	0x0C
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4	Event code	unsigned char	0 = Created/updated 1 = Deleted 2 = Rejected 3 = Delayed response
5	Network manager code	unsigned char	See Table 93
6	Service ID	unsigned char	0x00-0xFF
7	Service state	unsigned char	Bit 7: 0 = No service pending 1 = Service request pending Bits 6-0: 0x00 = Inactive 0x01 = Active 0x02 = Requested
8	Service flags	unsigned char	See Table 20
9	Application domain	unsigned char	0x00 =Publish 0x01 = Event 0x02 = Maintenance 0x03 = Block transfer
10-11	Destination Address	unsigned short	
12-15	Time	unsigned long	

Table 92 serviceIndication Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x0E
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_NO_RESOURCES

Table 93 Network Manager Codes

Code	Class	Description
0	Success	No command-specific errors.
4	Error	Passed parameter too small.
5	Error	Too few data bytes received.
6	Error	Device-specific command error.
8	Warning	Set to nearest possible value.
16	Error	Access restricted.
32	Error	Network manager is busy.
33	Error	Delayed response initiated-manager will attempt to add links.
35	Error	Delayed response: dead. This code is not used by Dust Networks managers. Refer to the manager's guide for further information.
36	Error	Delayed response: conflict. This code is not used by Dust Networks managers. Refer to the manager's guide for further information.
65	Error	Service request denied.
66	Error	Unknown service flag.
67	Error	Unknown application domain.
68	Error	Unknown nickname.

events

The *events* command sends an event notification packet to the microprocessor informing it of new events that have occurred. The reported event is cleared from the mote when the mote receives acknowledgement in the form of a response packet from the microprocessor.

Table 94 events Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x0F
2	Length	unsigned char	0x09
3	Flags	unsigned char	See Table 1 Bit 0 = 0 (request)
4-7	New events	unsigned long	See Table 24
8	Mote state	unsigned char	See Table 33
9-12	Current alarms	unsigned long	SeeTable 34

Table 95 events Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x0F
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_NO_RESOURCES

dataReceived

The *dataReceived* command notifies the microprocessor that a packet was received. For details on best effort and reliable transport types, refer to “Sending Data to and Receiving Data from the Network” on page 7. When the microprocessor receives a reliable *dataReceived* request, in addition to acknowledging the request with a dataReceived response it must also respond using the *send* command (see “send” on page 48). Note that the *dataReceived* request is the only command that contains an extended header that adds four control bytes inside the packet.

Source address—If this is a response to an end-to-end reliable request (as indicated in the flags byte), the *source address* must be used in the *destination address* field in the response sent via the *send* command.

*Sequence numbe*r—The *sequence number* is the transport layer sequence number. If the data is an end-to-end reliable request (as indicated in the flags byte), the *sequence number* must be used in the response sent via the send command. If the data is not reliable, the *sequence number* will be zero.

Table 96 dataReceived Request

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x81
2	Length	unsigned char	Length of request
3	Flags	unsigned char	See Table 98 Bit 0 = 0 (request)
4-5	Source address	unsigned short	Address of packet source
6	Sequence number	unsigned char	
7	Packet length	unsigned char	
8	Reserved	unsigned char	0x00
9	Reserved	unsigned char	0x00
10-11	Reserved	unsigned char	0xFC12
12...8...	Data	Array of unsigned char	

Table 97 dataReceived Response

Msg Byte	Description	Data Type	Value
1	Cmd type	unsigned char	0x81
2	Length	unsigned char	0x00
3	Flags	unsigned char	See Table 1 Bit 0 = 1 (response)
4	Result	unsigned char	RC_OK RC_NO_RESOURCES

Table 98 dataReceived Request-Flag Bit Description

Bit	Description
7	Transport direction (valid only for Reliable transport type): 0 = End-to-end request 1 = End-to-end response
6	Transport type: 0 = Best effort 1 = Reliable (acknowledged)
5-2	Reserved (use 0's for these bits)
1-0	See Table 1

HDLC Packet Processing Examples

Example 1: Constructing an HDLC packet to send to the mote

This example demonstrates how to construct an HDLC packet for performing *setNVParameter<networkID>*. (All values are in hexadecimal.)

Step 1 Define HDLC packet payload:

Command type => 03

Length => 07

Flags => 02 (request packet to write both NV and RAM)

Reserved => 00 00 00 00

NVParameter ID => 03

Network ID => 00 7D

HDLC Packet Payload					
Cmd Type	**Length**	**Flags**	**Reserved**	**NVParam ID**	**Network ID**
03	07	02	00 00 00 00	03	00 7D

Step 2 Calculate FCS:

- Calculate the FCS using FCS-16 algorithm (RFC 1662) on the hexadecimal sequence 03 07 02 00 00 00 00 03 00 7D. The FCS (including 1's complement) is B2 9A.
- Append FCS to payload, FCS is sent least significant byte first (RFC 1662)

HDLC Packet Payload	FCS
03 07 02 00 00 00 00 03 00 7D	9A B2

Step 3 Perform byte stuffing.

To perform byte stuffing, check the HDLC packet payload and FCS for instances of “7D” or “7E” and replace as follows:

7D => 7D 5D

7E => 7D 5E

Note that the additional control bytes do not count against the message payload limit, as defined in “Request and Response Packet Format” on page 17.

HDLC Packet Payload (stuffed)	FCS (stuffed)
03 07 02 00 00 00 00 03 00 7D 5D	9A B2

Step 4 Add start and stop delimiters.

Enclose the above in start/stop flags (RFC 1662).

Note that some products may require an additional 7E start byte for high speed operation. Refer to the product datasheets for details.

Start Byte	HDLC Packet Payload (stuffed)	FCS (stuffed)	Stop Byte
7E	03 07 02 00 00 00 00 03 00 7D 5D	9A B2	7E

Or simply, the hexadecimal sequence:

7E 03 07 02 00 00 00 00 03 00 7D 5D 9A B2 7E

Example 2: Decoding an HDLC packet received from the mote

To understand how to decode an HDLC packet sent from the mote, let’s assume that the mote received a *getNVParameter<networkId>* command and replied with the following HDLC packet. (All values are in hexadecimal.)

Start Byte	HDLC Packet Payload (stuffed)	FCS (stuffed)	Stop Byte
7E	04 03 01 00 03 00 7D 5E	A2 91	7E

Step 1 (HDLC layer) strip off delimiters.

HDLC Packet Payload (stuffed)	FCS (stuffed)
04 03 01 00 03 00 **7D 5E**	A2 91

Step 2 Remove byte stuffing in the HDLC packet payload and FCS.

To remove byte stuffing, check for instances of “7D 5D” or “7D 5E” and replace as follows:

7D 5D=> 7D
7D 5E=> 7E

HDLC Packet Payload	FCS
04 03 01 00 03 00 **7E**	A2 91

Step 3 Confirm FCS.

Calculate the checksum for the HDLC payload.

HDLC Packet Payload
04 03 01 00 03 00 **7E**

Confirm that the FCS matches the FCS sent with the packet. Because the packet encodes FCS least significant byte first, in this example the calculated FCS should match “91 A2”.

Step 4 (Application layer) parse HDLC payload:

Since the flags value shows this is a response packet, it can be parsed into the following structure:

HDLC Packet Payload				
Cmd Type	**Length**	**Flags**	**Result**	**Message Payload**
04	03	01	00	03 00 7E

Command type 0x04 is a *getNVParameter*, so the first byte of the message payload is the NVParameter ID. The value of NVParameter ID is 0x03, so this is specifically, a response to *getNVParameter<networkId>*:

HDLC Packet Payload					
Cmd Type	**Length**	**Flags**	**Result**	**NVParam ID**	**Network ID**
04	03	01	00	03	00 7E

Therefore, the network ID payload of this mote is 00 7E. Because the packet encodes the payload least significant byte first, the calculated network ID in this example would be 7E 00.

Changing Network ID During SEARCH State

In most cases, the network ID should be set during the IDLE state before the issuing the join command. However, there are some instances where it is helpful to be able to change the network ID even after the join command has been issued. For example, if the network ID has been entered incorrectly and the mote is attempting to join, it will never join the network. The recommended course of action is to reset the mote and have it come up in a known, clean state. However, it is possible to use *setNVParameter<networkID>* (with the “update RAM” flag enabled) to change the network ID while searching to recover the unit. This second course of action has the following caveats:

- If you have a mote searching with an incorrect network ID and the mote has not heard any advertisements, then the mote has not started join exchange with the network manager. Changing the network ID in RAM in this situation will work.
- If you have a mote with a valid network ID and it has heard an advertisement and started to join the network manger, then changing the network ID to another value may disrupt the join process that is in progress.