BLUES WIRELESS

Notecard & Notehub API Reference

see also
http://github.com/note-arduino/examples for Arduino examples
http://github.com/blues repositories for other language bindings

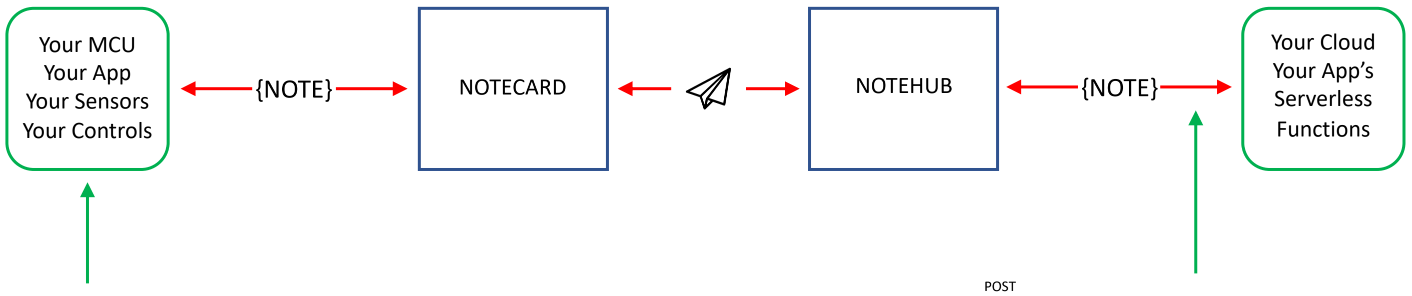
WARNING: Notecards prior to 1.2.3.10670 use “service.” rather than “hub.” as the prefix to Notehub-related requests. Use {“req”:”card.version”} to check for old firmware.

FUNCTIONAL OVERVIEW

In this document, when referring to the Notecard we are referring to the firmware that connects with a customer's MCU. This firmware implements a very simple synchronous protocol wherein \n terminated JSON-formatted commands result in JSON-formatted responses. All dialog between the MCU and the Notecard occurs using this JSON protocol.

The Notecard then communicates with an internet-deployed instance of a Notehub service. By default, Notecards are configured to communicate with the single-instance hyperscale SaaS service referred to as *notehub.io*, however the card can be configured to use a customer's own private Notehub service instances by using the "hub.set" JSON request.

The Notehub is designed to provide only a very thin layer of function: First, to auto-provision and enable customers to aggregate devices of similar function into "Fleets" whose operations can then be monitored, and, second, to configure such fleets with "Routes" by which JSON-formatted data arriving from devices are forwarded to a cloud app of your choosing.



```
setup()
  serial.println {"req":"service.set","product":"com.acme.airmon","minutes":60}
loop()
  serial.println {"req":"note.add","file":"air.qo","body":{"temp":72,"humid":61}}
```

```
POST
{"req":"note.add","file":"air.qo",
  "device":"imei:866425030050464","product":"com.acme.airmon",
  "when":1543380460,"where":"JMC2+67",
  "body":{"temp":72,"humid":61}}
```

In this simplest of examples, when a device starts up, it sends the Notecard a JSON request over its serial port to configure 1) the "Product UID" for this class of device, so that the service knows whose device this is, and 2) the maximum time that data should be allowed to stay on the device before it is sent to the service in a batch with other such staged data.

Then, as indicated by "loop()", without further setup, the device would then be able to issue a simple JSON command to add a JSON (a *Note*) to a collection of like-data (a *Notefile*). In this example the ".qo" in "air.qo" means "outbound queue" and "air" is just a name they made up. There's no need to pre-create the Notefile. Further, the Note's "body" is completely free-form JSON that is only ever interpreted by the customer's own app; there are no schema constraints.

When Notes were added using the "note.add" request, they were automatically augmented with time and location information if that info is available from the Notecard's hardware.

The Notecard then, using its own heuristics, will schedule connections to the Notehub service at which point it bidirectionally synchronizes notefiles as deemed necessary. The notes are moved or replicated from notefile staging areas between Notecard and Notehub storage.

Ultimately, the Notes are then forwarded to Routes as configured on the Notehub, as JSON, to the cloud services of the customer's choosing.

DEFINITIONS

Body A JSON object containing arbitrary application data, such as:

“body”:{“temp”:34.2,”alert”:true,”particle”:{“mass”:[5.1,12.32],”count”:[124,1800]}}

Payload A base64-encoded JSON string containing arbitrary application data, such as:

“payload”:”YWJjLS0tLS0tLS0tLS0tLXh5eg==”

Note A JSON object with both user data and a variety of metadata, such as:

{“when”:1543380460,”where”:”JMC2+67”,
“body”:{“temp”:34.2,”alert”:true,”particle”:{“mass”:[5.1,12.32],”count”:[124,1800]}}}

Notefile A named persistent JSON file containing an array of Notes, automatically created when the first note is added to it. Notefile names are arbitrary, but most typically indicate a sensor type or instance. The notefile extension determines certain attributes:

- .db is a database of JSON records that’s fully replicated between service & device, most commonly used for maintaining configuration and status information
- .qo is an outbound-only queue, with JSON records only being sent from device outbound to notehub, and wherein notes added on the device are automatically deleted once they’ve been successfully transferred
- .qi is an inbound-only queue, with JSON records only being sent from notehub inbound to the device, and wherein notes added on the notehub are automatically deleted once they’ve been successfully transferred
- .dbs, .qos, and .qis are *secure* variants of those respective types. Normally, communication sessions are unencrypted as a way of reducing bandwidth. For data that is more sensitive, using these notefile extensions will ensure that the data is always transferred on a TLS-encrypted communications session.

Request & Response A newline-delimited JSON object transmitted to and from the Notecard via the customer’s MCU, of this form. Each request will cause a single Response to be generated by the Notecard, and only a single request may be outstanding at any given time. If no error occurred during processing of a Request, the returned “err” field is “”.

{“req”:”*request-type*”, …*request-fields*…}
{“err”:”*reason-for-error*”, …*response-fields*…}

DEFINITIONS

Device A customer product containing a Notecard, or sometimes just the Notecard itself.

Device UID A globally-unique string identifier factory-assigned to the Notecard, such as the following. Note that this number is laser-engraved onto the Notecard, so that any customer wishing to use this as a part of their own product's serial number may easily do so as a part of their manufacturing process.
imei:866425030050464

Fleet A Notehub-unique entity containing a list of Devices - which can only belong to one Fleet.

Developer or **Customer** The entity accountable for development of the product within which the Notecard is embedded.

Account The metadata maintained by the Notehub for each person who has access to the Notehub and uses it for development or operations

Route A configuration setting that instructs Notehub where a given Device's Notes are to be transferred. A Route is generally an HTTP/HTTPS server, optionally with credentials.

Project (or "app") A Notehub entity belonging to a customer within which they manage:
- Accounts of the customer's designated fleet operators and developers
- Fleets that contain sets of auto-provisioned Devices
- Routes assigned to Fleets determining where Devices' Notes are transferred

Product UID (or Product ID) A Notehub-managed unique identifier that is used to match Devices with Projects. This string is used during a device's auto-provisioning to find the Notehub Project that, once provisioned, will securely manage the device and its data.

The format of a Product UID is "product:tld.company.uniquid". Once claimed by a customer on the Notehub, devices using "hub.set" with that given Product UID will automatically be assigned to that Team Project. For example,
product:com.tesla.* is how a company reserves its entire namespace on the Notehub
product.com.tesla.model-s.p85d is what their P85D model might use as its Product UID

Product SN (its Serial Number) An optional string assigned by a customer to a Device that enables them to correlate a given Device UID to a specific customer-manufactured product. The Product SN (and Product UID) may either be assigned to the Notecard using a Request during Notecard initialization by the host MCU, or may be programmed at the customer's point of manufacturing by sending that Request on its USB interface.

NOTECARD REQUESTS

The following sections will describe the various Notecard capabilities, as accessed through the Notecard's JSON Request/Response interface. As noted in the hardware documentation, this Request/Response interface is equivalently accessible through three different hardware ports.

From a software perspective, it's easiest and most straightforward to do Requests/Responses through the serial UART and USB ports, if use of those ports is convenient in the Product's hardware design. The simplest way to experiment with the Notecard and its JSON interface is to simply connect a USB cable from the Notecarrier's microUSB jack and your PC's USB port, and using a Windows or Mac terminal program such as CoolTerm configured at 9600/N/8/1.

Because the JSON commands are simple strings that are terminated by \n, standard serial "print" and "scan" libraries can be used to send Requests and receive Responses. A broad number of JSON-parsing libraries are available for any given MCU platform.

In some hardware designs, though, UARTS are at a premium, and so the Notecard also acts as an I2C slave with a very simple "one command" I2C interface enabling serial requests and responses to be sent and received over I2C. See I2C examples for further information.

The following sections describe the full breadth of the Notecard API's in order of how many features the MCU developer would like to take advantage of in their application.

For any given fields, *italic* means the field is optional.

For any given response,

> In all responses, an "err" field may be present in the returned object. If err is "" this means "no error", else it's a string indicating the reason for the error.
>
> Note that the "err" string is for developer consumption and is explicitly not intended for display to end-users. For future-proof programmatic error handling, brace-delimited keywords should be checked – for example, by searching for the string "{connection-failure}" within the err field in this sample response:
>
> {"err":"sync: unable to check for changes: {connection-failure} service disconnect"}

Note that part of the flexibility of JSON is the fact fields can be added without impacting applications that are written just to examine fields that are expected. When looking at the documentation of functions below, it must be anticipated by an application that future versions of the Notecard may extend those requests with new JSON fields in the future, and that new JSON fields may be returned within the response structures. Any fields not currently anticipated must be ignored.

A Word about Our Use of JSON

There are certain behaviors that are fundamental to how the Notecard and Notehub use JSON:

- If a structure field is “” (string), 0 (int), or 0.0 (float), then that field may be omitted from all JSON structures including Requests, and that field will be omitted from JSON structures including Responses

- Representation of whole numbers will appear in JSON without a decimal point

- Integers greater than 2^31 and less than -2^31, while generally supported, are technically Undefined Behavior by the JSON specification and thus may be converted into floating point numbers by some implementations.

- Sending JSON from one part of the software or service to another, and getting that JSON back from the software or service, does not guarantee the order of fields nor the presence of any whitespace.

- Although tabs and CR and LF characters are technically allowable whitespace in JSON, they are not allowed in Notecard Requests or Bodies and will never be placed within Notecard Responses. The Notecard uses the “ndjson” technique of new line-delimited json. (see http://ndjson.org)

- The Notecard JSON libraries do support UTF-8 but there may be language representation limitations due to the constraints of JSON within embedded systems.

1 **Essential Requests**

By far the most prevalent firmware design pattern is one in which the Device first initializes and then enters some form of loop in which data values are sensed and then sent to the cloud.

Periodic automatic connection to the cloud is the most natural, but manual may also be used.

Finally, most apps will find it useful to retrieve the values Notehub-administered Device-, Fleet-, or Project-scoped environment variables, so as to centrally tune Device behaviors.

1.1 **hub.set** sets card to use periodic vs continuous cellular connections

{"req":"hub.set"
    “product”:”notehubRegisteredProductUID”
    “sn”:”developerProductSN” is the end product’s serial number
    “mode”:
      ”periodic” is default mode which periodically connects to the Notehub
      ”continuous” for high-power devices with always-on cellular; data still syncs periodically
      ”minimum” disables automatic connection to service (see hub.sync)
      “off” disables both automatic and manual connection to the service
    “minutes”:maxWaitToSyncOutboundData such as newly added notes that may be queued
    “hours”:maxWaitToSyncInboundData such as environment variables from the service
    “sync”:autoSyncIfContinuous if we are in continuous mode, automatically and
        immediately ‘sync’ every time a change is detected to a notefile on the notehub
} → {}

1.2 **note.add** adds a note to an outbound queue notefile, creating it if it doesn’t yet exist

{"req":"note.add"
    “file”:”queueNotefileID” the NotefileID ending in “.qo” or “.qos” (default: “data.qo”)
    “body”:{optionalBodyJSON} (if no body OR payload, just ensures that the file is created)
    “payload”:”optionalBinaryPayloadB64==”
    “sync”:boolForceUploadNowBecauseOfUrgency
} → {
    “total”:numberOfNotesInFile
}

1.3 **env.get** return a service-configured environment variable

{"req":"env.get"
    “name”:”variableName” case-insensitive. If “”, the entire set of active vars is returned.
} → {
    “text”:”value” if a name was specified
    “body”:{“name”:”value”,”name”:”value”,…} if no name was specified
}

1.4 **env.default** sets default values for a local host MCU variable. This value is returned by env.get if neither env.set has overridden the variable nor anything on the notehub has overridden the variable.

{"req":"env.default"

“name”:”variableName” case-insensitive.

“text”:”value” is the value, or if “” (or omitted) the variable is deleted

} → {}

1.5 **env.set** sets the value for a local host MCU variable. This value is returned by env.get and cannot be overridden by the notehub.

{"req":"env.set"

“name”:”variableName” case-insensitive.

“text”:”value” is the value, or if “” (or omitted) the variable is deleted

} → {}

1.6 **hub.get** retrieves current service configuration parameters

{"req":"hub.get"} → {

“device”:”DeviceUIDForNotehub”

“host”:”NotehubOrWebHost”

“product”:”ProductUIDForNotehub”

“sn”:”ProductSNForNotehub”

}

1.7 **hub.status** displays whether card is in connected, disconnected, or intermediate states

{"req":"hub.status"} → {

“status”:”currentNetworkConnectedStatus”

“connected”:trueIfConnected if Notehub or Web host is online and connected continuously

”count” is number of bars of cellular coverage, when connecting or connected

}

1.8 **hub.sync** manually initiates synchronization with the service, beyond just periodic sync

{"req":"hub.sync"} → {}

1.9 **hub.sync.status** displays information about the last synchronization with the service

{"req":"hub.sync.status"

“sync”:trueIfshouldSyncBeAutoInitiatedIfPendingOutboundData

} → {

“status”:”previousSyncStatus” or “” if the previous sync completed successfully

“time”:unixEpochTimeOfLastSyncCompletion

“completed”:numOfSecsSinceLastSyncCompletion

“requested”:numOfSecsSinceLastExplicitSyncRequest

“alert”:trueIfErrorOnLastSync

}

1.10 **hub.log** enqueues a “device health” message to the service admin

{"req":"hub.log"

“text”:”message” enqueues a text message to the service admin in _health.qo

“alert”:trueIfAnUrgentCondition

“sync”:trueIfSyncShouldBeImmediatelyInitiated

} → {}

2 Time and Location Requests

By default, the Notecard module's GPS function is off and it has no sense of location. The same is true of time, until it connects to either the cellular network or GPS. Each note created or updated is tagged with the then-known time and location.

2.1 **card.time** retrieves the current time/date, or returns 0 for time if not yet available

{"req":"card.time"} → {
    "time":unixEpochTime
    "zone":"abbrev,full" local time zone (if cell tower is recognized)
    "minutes":numberOfMinutesEastOfGMT (if cell tower is recognized)
    "country":"US" (if cell tower is recognized)
}

2.2 **card.location** retrieves the current location, or returns an error if not yet available

{"req":"card.location"} → {
    "lat", "lon" indicate current location
    "time":unixEpochTime if location is valid, the time when this GPS location was sampled
    "area": "inside" or "outside" if ringfence is enabled
    "max":metersFromRingfenceCenter if ringfence is enabled
    "status":"locationUpdateStatusInformation"
}

2.3 **card.location.mode** sets and retrieves current location-related config settings

{"req":"card.location.mode"
    "mode": "" to retrieve the current mode and status, or:
      "off" mode turns off module's GPS function (the default) to minimize power draw
      "periodic" mode enables and samples the GPS every "seconds" if the device has moved
      "continuous" mode keeps GPS powered always-on, updating location continuously
    "seconds":secs if "periodic" mode, the GPS will be sampled at this interval
    *"vseconds":voltageModeString*** optionally overrides "seconds" w/voltage-variable value
    "delete":true to delete the last known location stored in the module
    "max":metersFromRingfenceCenter to enable ringfence (lat & lon must specify center)
    "minutes":ringfenceDebounceMins to debounce ringfence events; else 5m is the default
    "lat" & "lon":inDegrees sets center of circular ringfence (not specified uses current loc)
} → {}

2.4 **card.location.track** sets and retrieves current location tracking-related config settings, which are only relevant when the location mode is "periodic".

{"req":"card.location.track"
    "start":true starts recording a GPS track in notefile
    "heartbeat":true if "start" & this are both true, also include data when there's no motion
    "stop":true stops recording a GPS track in notefile
    "hours":hrs If "heartbeat" is true, add heartbeat at this interval rather than regular period
    "file":"trackNotefileID" to change track notefile name from the default "_track.qo"
} → {}

ADVANCED: INBOUND REQUESTS

Although most CIoT applications rely predominantly on upstream data communications and on environment variables for configuration, as described as Essential Requests, more advanced applications might find the need for having small databases of records shared with a cloud service, or server-based inbound command and control.

These databases might contain device configuration information, sensor information, state information, performance information and so on. The two key aspects of databases over queues are 1) that their records have keys called Note IDs that can be used to get/set their contents, and 2) that the database Notefiles are relatively stable and reasonable in terms of the number of records, so they can be loaded in memory when needed and won't don't overflow the device's Flash memory when stored.

For these applications, the Notecard provides an extensive set of Device-side and Notehub-side mechanisms to create, read, update, and delete records from databases. Databases are implemented as Notefiles which are fully replicated between Device and Notehub.

Also, as in the case of outbound queues for Essential Requests, inbound queues are also just implemented as Notefile.

The syntax of these database/queue requests is identical regardless of whether the commands are sent to the Notecard (for Device-side requests) or to the Notehub (for service-side requests). Just as it's easy to send JSON requests to the Notecard using any terminal program on a PC, requests can also be sent to the Notehub by using command line utilities such as CURL.

The Notehub API is an HTTP GET or POST, with the JSON request and response being passed in the body of what is posted and what is returned. The URL format of the request is as follows, where *notebox.net* is the domain of the Notehub serving the customer Device:

http://*notebox.net*?project="ProjectUID"&device="DeviceUID"

All requests below are supported on the Notecard, while requests supported via the Notehub HTTP API are noted as such with an indication of (GET) or (POST) as appropriate.

Change Trackers

When tracking inbound changes that may have occurred both within a notefile and across notefiles, it's convenient for efficiency that the app to be able to do so incrementally. As such, the Notecard implements the notion of a *change tracker*. To be clear - it's not required that apps use change trackers, but they're available when the app might need them.

Trackers are expressed as simply a user-generated text string, i.e. "new_inbound_requests", that the Notecard uses to track additions and deletions to notefiles. While tracking pending deletions, each notefile will contain an object called a *tombstone* for every deleted note that is being tracked by some tracker in that notefile. As such, if you're no longer using a tracker, be sure to delete it so that tombstones no longer need to be maintained for that tracker.

3 Database Requests

3.1 **note.add** adds a note, creating the notefile if it doesn't yet exist (HTTP POST also available)

```
{"req":"note.add"
    "file":"notefileID" is the NotefileID (default: "data.qo")
    "note":"noteID" if the notefileID has a ".db" extension, specifies a unique note ID
    "body":{optionalBodyJSON}
    "payload":"optionalBinaryPayloadB64=="
    "sync":boolForceUploadNowBecauseOfUrgency
} → {
    "total":numberOfNotesInFile
}
```

3.2 **note.update** updates a note in a .db by ID (HTTP POST also available)

```
{"req":"note.update"
    "file":"notefileID"
    "note":"noteID"
    "body":{bodyJSON}
    "payload":"optionalBinaryPayloadB64=="
} → {}
```

3.3 **note.delete** deletes a note from a .db by ID (HTTP POST also available)

```
{"req":"note.delete"
    "file":"notefileID"
    "note":"noteID"
} → {}
```

3.4 **note.get** retrieves a note (HTTP GET also available)

```
{"req":"note.get"
    "file":"notefileID" (default: "data.qi")
    "note":"noteID" (only to be used when file type is ".db")
    "delete":true in order to delete the note after getting it
    "deleted":true to allow retrieval of a deleted note
} → {
    "body":{bodyJSON}
    "payload":"payloadB64=="
}
```

3.5 **note.changes** incrementally retrieves changes within a specific notefile

```
{"req":"note.changes"
        “file”:”notefileID”
        “tracker”:”changeTrackerID”
        “max”:maxNumberOfNotesToBeReturned
        “start”:isTrackerToBeResetToBeginningBeforeGet
        “stop”:isTrackerToBeDeletedAfterGet
        “deleted”:shouldAnyDeletedNotesBeReturnedInTheseResults
        “delete”:areNotesReturnedInResponseToBeAutoDeleted
} → {
        “changes”:totalNumberOfPendingChangesInNotefileInclusiveOfThese
        “total”:numberOfTotalNotesInNotefile
        “notes”:{ “noteID”:{…},”noteID”: {…} ]
        (where the “…” per-file data structure has fields equivalent to “note.get” response)
}
```

3.6 **file.changes** performs individual or batch queries to see what’s new.

```
{"req":"file.changes"
        “tracker”:”changeTrackerIDtoTrackAcrossNotefiles”, omit to check totals across all files
        “files”:[ “notefileIDToFilterThatYouAreInterestedIn”, … ] filter results just to these
} → {
        “changes”:numberOfPendingChangesInNotefiles based on the tracker
        “total”:numberOfTotalNotesInNotefile
        “info”:{ notefileID:{…}, notefileID:{…} }
}
```

3.7 **file.changes.pending** returns info about changes are pending to be uploaded to notehub

```
{"req":"file.changes.pending"} → {
        “pending”:true if there are changes pending
        “changes”:numberOfPendingChangesAcrossAllNotefiles
        “info”:{ notefileID:{…}, notefileID:{…} }
}
```

3.8 **file.delete** deletes individual notefiles and all that they contain

```
{"req":"file.delete"
        “files”:[ “notefileIDToDelete”, … ]
} → {}
```

3.9 **file.stats** gets resource usage statistics about local notefiles

```
{"req":"file.stats"} → {
        “total”:totalNotes across all notefiles
        “changes”:totalChanges pending across all notefiles that need to be sync’ed
        “sync”:true if sufficient pending notes warrant a sync solely because of resource usage
}
```

4 Event Notification Requests

When developing an app “beyond the Essentials” that uses service-replicated databases and inbound queues, it becomes important for the host MCU to understand when inbound data becomes available and when various interesting events may transpire.

Since the Notecard operates as an independent peripheral, scheduling Notehub service connections on an asynchronous basis, the typical way that a developer might process inbound commands is to *poll* the Notecard periodically using *change trackers* to detect changes.

But this coding technique, without some further optimization, could be potentially quite expensive from a power perspective. As such, the Notecard exposes a hardware pin named ATTN which, as the name suggests, is intended to cause an interrupt on the host MCU when one or a number of events may occur on the Notecard. The app may the card.attn request several times to indicate what events it is interested in, such as specific files, or whether to be notified by connection or motion events. Then the app does an “arm” which brings the initially-high ATTN pin LOW. The ATTN pin will then go high on the earlier of an event or a timeout.

4.1 **card.attn** is used to sense events on a host configured for leading-edge interrupt detection
{"req":"card.attn"
- “mode”: is a comma-separated list of the following keywords
  - “” fetches the currently-pending events in files
  - ”arm” clear files events and causes ATTN pin to go LOW. The ATTN pin will then go HIGH (attn “fires”) on the earlier of an event occurring, or after seconds has elapsed. Note that if seconds is 0, there will be no timeout scheduled.
  - ”disarm” causes ATTN pin to go HIGH if it had been low
  - ”files” when armed, will cause attn to fire if any of files notefile IDs are modified, (or any file is modified if files is “”. Disable by using “-files”.
  - ”connected” when armed, will cause attn to fire whenever the module successfully connects to the service. Disable by using “-connected”.
  - ”motion” when armed, will cause attn to fire whenever the accelerometer detects module motion. Disable by using “-motion”.
  - ”location” when armed, will cause attn to fire whenever the GNSS hardware detects a valid location update of the module. Disable by using “-location”.
  - ”watchdog” is not an “arm” mode, but rather it will cause the attn pin to from HIGH to LOW then HIGH, if the notecard fails to receive *any* JSON request for *seconds*. When a host MCU’s power or NRST is wired to the ATTN pin, this can be used as a watchdog timer by the host MCU.
- *“files”:[“notefileID”, …]* the list of notefiles being watched (see “files” above)
- *“seconds”:numSeconds* is used for ATTN timeout when arming it, and by “watchdog”

} → {
- *“files”:[“notefileIDsOrEvents”, …]* All notefile IDs that have changed since last reset, and also the keywords for all events that have occurred. Keywords for events are:
  - “timeout” if no events occurred at all within “seconds” after reset
  - “files” if files have been modified since the reset
  - “connected” if the module connected to the service since the reset

```
                    “motion” if module motion has been detected since the reset
}
```

5 Web Service Requests

In some cases when developing an app that goes "beyond the Essentials", it can be useful for the module to do some small but critical web transactions to a service other than the Notehub, getting the response from the web service in real-time.

To enable these small transactions to occur, the Notehub can act as a web proxy server that executes JSON web transactions. On the Notehub, an admin first configures a Route that maps to a specific web server, along with optional authentication information. The admin assigns this route an "alias" – a brief name used below that uniquely refers to that Route.

The customer's MCU then temporarily places the module online using the "continuous" service mode, and performs a transaction. Both the body and response must be JSON, and the request or response may carry a binary payload. On the service, the body and payload are carried within a JSON data structure that additionally provides the Notehub-authenticated device UID and SN.

This web proxying methodology is powerful in that it enables the called service to leverage the Notehub's secure authentication, and allows the development of customer firmware that needn't contain hard-wired URLs, authentication information, or cryptographic keys, all of which likely will change over the lifetime of a customer device.

5.1 **hub.set** temporarily establishes continuous connection to the notehub

```
{"req":"hub.set"
        "mode":"continuous"
        "minutes": maxWaitToSyncOutboundData such as newly added notes that may be queued
        "hours": maxWaitToSyncInboundData such as environment variables from the service
        "align":true to align syncs onto a regular time-periodic cycle vs being "maxWait"-triggered
} → {}
```

5.2 **web.get** or **web.put** or **web.post** performs a simple HTTP/HTTPS JSON transaction

```
{"req":"web.get" or "web.put" or "web.post"
        "route":"notehubRouteAliasForWebServer" alias of a Web Server Route on the Notehub
        "name":"optionalWebPageRelativeToHost" such as "/getStatus" or "/doTransaction"
        "body":{JSONRequest} if web.put or web.post
} → {
        "result":HTTPResultCode
        "body":{JSONResponse}
}
```

The Notecard itself is designed to be battery operated, with extremely low current consumption. As a result, in many applications it is the host MCU and its peripherals, not the Notecard, that can be the primary power drain in a battery-powered product.

The notecard implements two tools that can be useful to a product designer and its firmware developer. The first is simply a technique that we recommend be used by the firmware designer in order to cause the Notecard's modem to only be active as a function of how much energy is available. The second is a design technique wherein the product designer can arrange to have the Notecard shut down the host MCU in its entirety, for dramatic power savings.

6 Modem Power Management Requests

In battery-operated systems – particularly those using energy harvesting technologies such as solar power – the available energy rises and falls between quite some extremes. The Notecard, while being extremely low-power in its idle state, and while being extremely low-power in its ability to *stage* data that the app adds to notefiles using note.add, can use a nontrivial amount energy when the modem is transmitting to or receiving from the cellular network.

It's therefore a best practice for apps to vary the periodic service connection intervals based upon the available energy as detected by the host's use of a *fuel gauge* chip or PMIC. For example, when there is ample energy available, one might configure the Notecard to synchronize outbound data every hour while processing inbound every 6 hours. On the other hand, when the energy level is low, the firmware might configure the Notecard to stage for a much longer time, synchronizing outbound and inbound only once every 24 or 48 hours.

In extreme cases, when the app senses that the battery is so low that it might cause damage to its cells, it's important that the app have the ability to tell the Notecard "under NO circumstances do I want you to turn the modem on". This is more or less like *airplane mode*.

6.1 **hub.set** parameters used to dynamically throttle modem activity

```
{"req":"hub.set"
        "mode":
          "periodic" is default mode which periodically connects to the Notehub
          "off" prevents the modem from turning on under any circumstances
        "minutes": maxWaitToSyncOutboundData such as newly added notes that may be queued
        "vminutes":voltageModeString** optionally overrides "minutes" w/voltage-variable value
        "hours": maxWaitToSyncInboundData such as environment variables from the service
        "vhours":voltageModeString** optionally overrides "hours" w/voltage-variable value
} → {}
```

7 Host Power Management Requests

In some cases, the host MCU and its software cause so much current to be drawn that it is quite difficult to achieve a low power system design. In certain cases, however, it may be possible to design a system in a way that uses the Notecard to place the host MCU into a state in which it is completely off – only drawing the iQ of the MCU's power regulators when in a shutdown state.

The Notecard can be used both as a timer and also, provided the host MCU lacks its own NVM, as a place to temporarily hold state that would have been kept in the MCU's RAM.

This is made possible by using the Notecard's ATTN pin, which is pinned HIGH at boot and which stays HIGH until card.attn is told to reset. Thus, if the hardware designer connects ATTN to the EN pin on the MCU's regulator, this will achieve the desired results. Upon boot, the host MCU should then always use a different form of the card.attn request to determine whether this is an initial boot or whether it is being awakened from sleep.

7.1 card.attn can be used to place the host MCU into a power-off sleep state
{"req":"card.attn"
    "mode":
      "reset" clear causes ATTN pin to go LOW, which will go HIGH after *seconds* has elapsed
    "seconds":numSeconds to sleep
    "payload":"payloadB64==" *is used by reset to specify an optional payload of data to be*
        held in memory, which may later be retrieved/cleared picking it up via *start*. This
        payload must be retrieved within a 120s window before/after *seconds* has elapsed.
} → {}

7.2 card.attn can be called at boot to restore payload and determine if awakened from sleep
{"req":"card.attn"
    "start":true to pick up the payload
} → {
    "payload":"payloadB64==" if payload is being retrieved and cleared by *"start":true*
    "time": unixEpochTime is the current time
}

8 Voltage Monitoring
In some hardware designs, the Notecard's V+ input will be connected directly to a battery. This can be advantageous because it's the best way of ensuring that any potential current spikes during wireless transmission served by the large "tank" of energy in the battery. The extremely low standby power draw of the Notecard also minimizes potential downsides of such a design.

8.1 card.voltage measures the current V+ voltage level and provides historical voltage trends

```
{"req":"card.voltage"
        "hours":numHours to analyze, up to 30 days (defaults to as much is available)
        "offset":numHours to go back into the past before starting analysis (defaults to 0)
        "vmax":voltageLevel above which measurements for a period are ignored
        "vmin":voltageLevel below which measurements for a period are ignored
                        If vmax and vmin are 0, they're set to 4.5 and 2.5 respectively which
                        is effective for LiPo batteries, ignoring USB charging & "off" periods.
} → {
        "value":currentVoltage
        "hours":numHours for which data was available for trend analysis
        "vmin":voltageLevel, the lowest value during the measured period
        "vmax":voltageLevel, the highest value during the measured period
        "vavg":voltageLevel, the average value during the measured period
        "daily":voltageChange, the change of moving average over the past 24 hours
        "weekly":voltageChange, the change of moving average during the past 7 days
        "monthly":voltageChange, the change of moving average over the past 30 days
}
```

Temperature Monitoring
Sometimes a product can be deployed in environments that are so cold or so warm as to affect battery performance. In these situations, it can be useful for a developer to be aware of the temperature within the electronics enclosure. The Notecard has a calibrated absolute temperature sensor that can be used to measure the temperature around the Notecard.

8.2 card.temp measures the current temperature

```
{"req":"card.temp"} → {
        "value":currentTemperatureDegreesCentigrade
}
```

Motion Monitoring
In some cases, a host MCU may have good reason to change its behavior when the product is in motion or in an incorrect physical orientation. It may also wish to take actions based on the notecard's motion.

8.3 card.motion returns information about the physical motion characteristics of the Notecard
{"req":"card.motion"
    "minutes":amountOfTime to sample for buckets of movement
} → {
    "count":numberOfMotionEventsSinceLastCalled
    "alert":trueIfFreeFallSensedSinceLastCalled
    "time":unixEpochTimeOfLastMotionEvent
    "status":keywordList which is a comma-separated list of event keywords including
        "face-up", "face-down", "portrait-up", "portrait-down",
        "landscape-right", "landscape-left"
    "movements":"movementBuckets" up to 250 samples of movement, with a single
        "bucket" per character, showing most-recent to least-recent the number of
        movements sensed in that bucket of time. The movement is indicated "base 36"
        with 0-9 A-Z and then * if the number of movements is greater than 35.
}

8.4 card.motion.mode configures motion monitoring
{"req":"card.motion.mode"
    "stop":trueToStopMonitoringMotion
    "start":trueToResumeMonitoringMotion
    "seconds":periodForEachBucket of movements to be accumulated
    "sensitivity":level of accelerometer, as follows (1 is default)
        1 for 7.808G (lowest sensitivity)
        2 for 3.904G
        3 for 1.952G
        4 for 0.976G
        5 for 0.244G (highest sensitivity)
} → {}

8.5 card.motion.track configures automatic tracking of movement
{"req":"card.motion.track"} → {
    "start":true starts recording a motion track in notefile
    "stop":true stops recording a motion track in notefile
    "minutes":mins the maximum period at which notes will be recorded in track
    "count":numBuckets the number of most recent motion buckets to examine
    "threshold":numBuckets the # of buckets of count that must indicate motion to track
    "file":"trackNotefileID" to change track notefile name from the default "_motion.qo"
}

8.6 card.motion.sync configures automatic sync triggered by movement
{"req":"card.motion.sync"} → {

"start":true starts motion-triggered syncing
"stop":true stops motion-triggered syncing
"minutes":mins the maximum frequency at which sync will be triggered
"count":numBuckets the number of most recent motion buckets to examine
"threshold":numBuckets the # of buckets of count that must indicate motion to sync

}

Voltage-Variable Behaviors

When optimizing the behavior of battery-operated devices, it can be important to vary the timing of various operating parameters such as upload interval or sensor measurements based upon the level of available energy. If Voltage is used as a proxy for available energy, Voltage Mode strings can be crafted to easily set the voltage based on the current mode. The default voltageModeString placed in the "mode" variable is blank (because we don't know how you will power the Notecard), but for LiPo a typical string to use would be "max:4.6;high:4.0;normal:3.5;low:3.1;dead:0" which means that above 4.6V the mode will be "max", 4.0-4.6 will be "high", and so on. This string can be customized for differing battery chemistries by just altering this "mode" field. The current mode (such as "normal") will be returned in the "mode" output field.

A host application may have an application whose sensor timings may be based on the current voltage. Furthermore, that host application may wish an administrator to override those timings with an environment variable. This is easy to do using the "name", "value", and "vvalue" fields.

8.7 card.voltage measures the current V+ voltage level and provides historical voltage trends
{"req":"card.voltage"

*"mode":*voltageModeString used for customizing full, normal, low, and dead thresholds
"name":"temp_sensor_mins" is the name of an environment variable that will be used for a given sensor's administrative override of sensor timings
"vvalue":"max:15;high:30;normal:60;low:240;0" is an example of the voltage-variable values to be returned in the value field based on the current voltage level

} → {

"mode":currentVoltageMode based on current battery voltage
"value":60 is the value that is returned if the vvalue field above were used and the current voltage mode is normal

}

9 Data Usage Optimization Requests

In order to give extreme priority to developer flexibility and ease of use, the Notecard's API allows the developer to use the entire richness of JSON in the body of every Note, and it does not require any consistency among notes within any given notefile.

The design of the notefile system is that it is primarily "memory-based", designed to operate within the range of 100 notes per notefile. But some apps by their very nature, such as GPS tracking applications, need to record, and stage for communications, bursts of data that may change every several seconds for limited periods of time – staging far more than these limits would allow. For these types of applications, we provide a means to use a "flash-based" storage system in order to optimize memory, storage, and communications bandwidth.

By supplying a "JSON body template" within any given .qo or .qos notefile, the Notecard enables higher-volume direct-to-flash enqueueing of highly-structured bulk data. This JSON template acts as a hint enabling the notecard to store & compress the data internally as simple fixed-length records rather than storing it with full flexibility of JSON. Beyond a single note.add request to put the template into effect, no other API changes are necessary to take advantage of this optimization.

For example, a developer might use a template that is set to {"temp":1.1, "count":1, "alert":true, "status":"AAAAA"}. By using this hint, the Notecard may then assume that each note in the notefile will be stored with only these four fields, the temp will be stored as a floating point number, the count will be stored as an integer, the alert will be stored as a true/false flag, and status will contain a string up to a maximum of 5 characters.

When a notefile template is in effect for a given notefile, note.add request body and payload are verified for conformance. For notes added when the template is in effect, data will be stored and transmitted in a manner that is fixed size and binary-compressed. A new template may be set at any time without having any impact on notes that had been previously enqueued.

9.1 note.template optimizes efficiency with a note format "hint"

{"req":"note.template"

    "file":"notefileID"

    "body":{sampleData} not present to clear the template, or a template body of the form:

      "myBoolField":true where the presence of true indicates bool

      "myIntegerField":1 where the *specific* value 1 indicates a 32-bit signed integer
      (Advanced: specifying 11, 12, 13, 14, or 18 will use a 1, 2, 3, 4, or 8 byte signed integer)

      "myFloatField":1.1 where the specific value 1.1 indicates an 8-byte double float
      (Advanced: specifying 12.1, 14.1, or 18.1 will use a IEEE 754 floats of 2, 4, or 8 bytes)

      "myStringField":"aaaaa" a repeated string of "a" whose length is max length of string
      (Advanced: a numeric string like "23" indicates a that the field's max length is 23 chars)

    "length":payloadLength is the maximum length of the binary payload to be sent

} → {

      "bytes":numBytesThatWillBeTransmittedPerNoteBeforeCompression

}

10 Data Usage Measurement Requests

Integral to the Notecard is that it may only transmit and receive a certain quantity of data over its lifetime. The amount of data that is part of a given Notecard, and its projected physical and logical lifetimes, are specific to the card and associated offers.

The data transmitted and received by the Notecard is proportional to the amount of *user data* supplied at, for example, a note.add. However, in some cases it may be more than that (because of, for example, per-session TLS and TCP overhead), and in other cases it may be much less than that (because of compression).

Because it is up to the developer as to how often they generate data, and how large that data may be, the Notecard contains tools enabling the developer to test the actual use of data so that they may compute projections of Notecard lifetime based on their specific workload.

10.1 card.usage.get returns the card's actual network usage statistics

```
{"req":"card.usage.get"
      "mode":
        "total" returns the stats since the card was first activated
        "1hour" or "1day" or "30day" returns that unit of stats, skipping back by offset units
      "offset":numberOfUnitsToSkipBackward
} → {
      "seconds":numberOfSecondsElapsedSinceCardWasFirstActivated
      "time":epochTimeOfStatsBase
      "bytes_sent":numberOfBytesSent
      "bytes_received":numberOfBytesReceived
      "notes_sent":numberOfNotesSent
      "notes_received":numberOfNotesReceived
      "sessions_standard":numberOfStandardHubSessions
      "sessions_secure":numberOfSecureHubSessions
}
```

10.2 card.usage.test projects lifetime at current use rate, given theoretical quota MB available. If no arguments are supplied for days and hours, all data since activation is used.

```
{"req":"card.usage.test"
      "days":mostRecentNumDaysOfActualUsageToUse
      "hours":mostRecentNumHoursOfActualUsageToUse
      "megabytes":numberMBOfQuotaToUse if 0, defaults to 1024 (1GB)
} → {
      "bytes_per_day":averageDataRateOverPeriod
      "days":numberOfDaysActuallyUsedForLifetimeTest
      "max":maximumNumberOfDaysOfLifetimeForLifetimeTest
      (Also, same fields returned as card.usage.get so you can see the usage rate over period.)
}
```

11 Host Firmware Update Requests

The host MCU that's used to send requests the Notecard may be of any type and from any manufacturer, and so it is not possible for the Notecard to offer an entire end-to-end firmware update capability in an automated processor-independent manner.

That said, the ability to update its firmware *over the air* is an important capability for all microcontrollers, and so we have externalized portions of the Notecard's own protocols so as to streamline firmware update data flows for the MCU developer. These mechanisms can eliminate a significant part of the burden of implementing firmware update for the host MCU.

The general flow of firmware update begins using the Notehub's device management interface:

1. A Project administrator uploads a binary firmware file into their project, and configures its metadata including a name and a verification string that should be inside the binary that will clearly expose its version number. No special packaging tools are necessary prior to upload. The Notehub back-end automatically generates an MD5 hash of the firmware and displays it for administrator verification.
2. The Project administrator sets the values of several special "firmware update" environment variables on either a Fleet or Device basis. These variables declare to those Notecard that they should be running a specific version of the firmware.
3. Over the next hours or days, the Notecards which sense that they need to install new firmware will download it progressively in the background. The Notecard places the download into a special "firmware storage" area that is larger than normally available to the Notecard, and which takes more power to access. Once downloaded into firmware storage, the Notecard will verify the MD5 hash to provide download assurance.
4. At a time appropriate for the developer's app, the developer will use the dfu.status request to see that DFU data has been fully downloaded and is ready for processing, and to retrieve the metadata of the downloaded firmware.
5. The MCU app will then reboot into its own custom bootloader that will use hub.set to place the Notecard into a mode where it can access firmware storage, then repeatedly using dfu.get to load chunks of the firmware binary into its own memory. Once fully downloaded, it will re-flash and restart the MCU.

The requests involved in background download of developer firmware are as follows.

11.1 **dfu.status** gets/sets the background download status of firmware for the developer's MCU

{"req":"dfu.status"
    "stop":true to optionally clear DFU state and delete any local firmware image
    "status":"reasonWhyStopped" is a message that can indicate to the service why
    "version":"hostFirmwareVersion" to notify the Notehub admin what firmware is running
    "vvalue":"aVoltageVariableString" to control whether or not DFU is to be enabled
} → {
  "mode":
    "idle" no DFU download in progress and no data previously downloaded for DFU
    "error" download or verification failed and process is halted; status has reason
    "downloading" if it's in progress; status has info about progress
    "ready" if DFU data is fully downloaded and verified
  "status":"currentStatusOfDFUDownload"
  "body":{uploadedFileInfo} is returned only when mode is "ready". The file info contains:
    {
    "length" is the length of the downloaded firmware
    "md5" is a hexadecimal string containing the (verified) MD5 hash of the firmware image
    "crc32" is the IEEE 32-bit CRC of the firmware image
    "found" is a string that is automatically extracted from the firmware image when it was uploaded, generally indicating the version number of the image. The Notehub admin uploading the image optionally may supply a partial string to search for such as "Airnote Version ", and the service extracts the full string containing this.
    "info" is an optional JSON object specified by the Notehub admin when the image is uploaded, containing any arbitrary metadata that may be useful to the developer
    }
}

11.2 **hub.set** Enters "DFU Mode", halting communications activity and establishing an internal connection to a local bulk storage area reserved for downloaded firmware. After firmware has been retrieved with dfu.get, use hub.set to change back to the desired operational mode such as periodic.

{"req":"hub.set"
  "mode":"dfu"
} → {}

11.3 **dfu.get** (available in DFU mode only) retrieves downloaded firmware data

{"req":"dfu.get"
  "offset":"offsetWithinFile" is the 0-based offset from which to read
  "length":"numberOfBytesToRead" may be specified as 0 to check for status of pending mode switch to DFU Mode, which may be delayed because of in-progress comms.
} → {
  "payload":"payloadB64==" is a block of data of requested length

```
}
```

12 Notecard Maintenance Requests

12.1 **card.version** returns version information about the module

```
{"req":"card.version"} → {
        "name":"notecardBrandAndModel" for display
        "version":"notecardVersion" of firmware and hardware
        "value":notecardVersionMajor.Minor for ease of checking programmatically
}
```

12.2 **card.status** returns general information about the module's operating status

```
{"req":"card.status"} → {
        "status":"generalStatusInformation"
        "usb":trueIfNotecardIsBeingPoweredByUSB
        "storage":pecentageOfStorageUsed
        "time":unixEpochTime when Notecard first obtained the time after starting up
        "connected":trueIfConnected if Notehub or Web host is online and connected
}
```

12.3 **card.restart** performs a firmware restart of the module

```
{"req":"card.restart"} → {}
```

12.4. **card.restore** reformats the module to factory-initialized conditions and restarts

```
{"req":"card.restore"
        "delete":areNotecardConfigSettingsAlsoToBeDeleted
} → {}
```

13 Private Notehub Service Configuration Request

13.1 **hub.set** sets card to use a private notehub

```
{"req":"hub.set"
        "host":"customNotehubHostDomain" or "-" for default notehub.io host
} → {}
```

14 Advanced Card Configuration Requests

14.1 **card.wireless** can be used to customize the behavior of the modem, or to view the last known network state

```
{"req":"card.wireless"
        ”mode”:
          ”-” reset mode configuration to default
          ”auto” select automatic band scan mode (tries M1, NB1, and EGPRS)
          ”m”, “nb”, “gprs” restrict selection to Cat-M1, Cat-NB1, or EGPRS
          “xxx” use custom connection string xxx with APN, bands, seek order, …
} → {
        “status”:currentStatusOfWireless
        ”net”:{
          “rssi”:receivedSignalStrength
          “rsrq”:receivedSignalQuality
          …
          “updated”:timeOfLastSignalQualityUpdate
        }
        ”count”:numberOfBarsOfSignalQuality of the values in “net”
}
```

14.2 **card.io** can be used to modify card’s I2C address

```
{"req":"card.io"
        ”i2c”:alternateAddress or -1 to reset to default address. This is a decimal number.
} → {}
```

14.5 **card.contact** can be used to set or retrieve info about designated card maintainer

```
{"req":"card.contact"
        ”name”:”mainterName”
        ”org”:”mainterAffiliation”
        ”role”:”mainterRole”
        ”email”:”mainterEmail”
} → {
        ”name”:”mainterName”
        ”org”:”mainterAffiliation”
        ”role”:”mainterRole”
        ”email”:”mainterEmail”
}
```

15 AUX Pin Configuration Requests

Both the USB port and the AUX-RX/AUX-TX ports may be used to display the continuous trace log of information that is generated by the Notecard. When connected using a a terminal emulator, you may enter either standard JSON requests or special tracing commands, both ending in <enter>. The "t" command enables tracing, enabling a developer to better understand how the Notecard works by enabling them to observe its operations. Trace options are available by using the "?" command.

In addition to tracing, the **card.aux** request can be used to configure various uses of the general purpose AUX1-AUX4 and AUX-EN/RX/TX pins.

15.1 **OFF Mode** (default)

When not using any special AUX mode, this ensures that signals on the pins are ignored.

{"req":"card.aux", "mode":"off"} → {}

15.2 **AUX TRACK Mode**

In this mode, the AUX1-AUX4 pins are used for purposes specifically related to GPS Tracking operations, in order to interface with external hardware related to that tracking.

In track mode, the AUX1-AUX4 pins are configured as follows:

> AUX1 is an active-low "button" input control. When the button is activated, it should be wired to pull the AUX1 pin low to GND. This button simply adds a binary "button pushed" event to the tracking database, and initiates an immediate cloud sync.
>
> AUX2 is an active-high "event occurred" input control. Whenever the leading edge of a pulse is seen on the AUX2 pin, a counter is incremented that is added to the tracking database whenever the next tracking event or heartbeat occurs.
>
> AUX3 is an active-high "event occurred while in motion" input control. Whenever the leading edge of a pulse is seen on the AUX3 pin AND the device is actively in-motion, a counter is incremented that is added to the tracking database whenever the next tracking event or heartbeat occurs.
>
> AUX4 is an "in-motion" output signal. Whenever the device has been in motion for at least several seconds, AUX4 is set high. After the device has seen to have settled down and is no longer in-motion, the output signal is brought LOW.

{"req":"card.aux", "mode":"track"} → {}

15.3 **AUX GPIO Mode**

In this mode, the AUX1-AUX4 pins are used for general purpose I/O, and the AUX-EN/AUX-TX/AUX-RX pins are used for card behavior tracing during software development using a UART-to-USB adapter (e.g. the FTDI TTL-232R-RPi) and a terminal program such as (e.g. “screen” or “CoolTerm”) configured for 115200 N81. Note that AUX-EN must be set HIGH for tracing to be enabled.

```
{"req":"card.aux"
    ”mode”:”gpio”
     “usage”:[”modeForAUX1”, ”modeForAUX2”, ”modeForAUX3”, ”modeForAUX4”]
      “” to leave this pin’s mode unchanged
      “off” (default) to disable the pin
      ”high” to set pin to output mode and to set its value to HIGH
      ”low” to set pin to output mode and to set its value to LOW
      ”input” to set pin to input mode
      ”count” to set pin to input leading-edge pulse counting mode
    When using count, all must be configured at once and are set with:
    “seconds”:numOfSecondsPerSample (0 will total into a single sample)
    “max”:maxNumOfSamples (excess counts are added to final sample)
    “start”:trueToResetAllCountersNow
} → {
    ”state”:[{“high”:trueOrFalse,”count”:[countsEachSample]},…] Indicates the current pin
        state and counter values for each of the AUX pins
    “time”:unixEpochTime indicates time when counting started, if any count is enabled
    “seconds”:numOfSecondsPerSample if any count is enabled
}
```

15.4 **AUX Monitor Mode**

This mode enables the AUX pins to be used to implement the basic functions to be placed on the faceplate of a device so that it may be used to monitor and test Notecard activity. It also allows a host MCU to selectively override the functions of LEDs with its own error status codes.

In monitor mode, the AUX1 pin is configured as a “test button”, and the AUX2-AUX4 pins are configured as outputs for LED control.

AUX1 is the “COMM TEST” button, an “active low with pullup” input pin that is wired to a momentary contact switch that is Normally Open, and pulled to GND when pressed. The function of this button is referred to as “COMM TEST”, and by pressing it a test note is added to _health.qo and a manual sync process is initiated.

AUX2-AUX4: are multifunction LEDS that should ideally be the specific color below, so that their function may be uniformly described across products. These LEDs may be connected directly to the GPIOs, with resistors chosen to ensure a maximum current draw of 20mA per LED.

AUX3 is a GREEN LED referred to as “COMM BUSY”. The LED turns on and displays a pattern based on communications status.

AUX4 is a RED LED referred to as “COMM ERROR”. The LED turns on continuously for 2 minutes after there is a failure to connect to cellular. To conserve battery life, after the first 2 minutes it only turns on for about 4 seconds every minute.

AUX2 is a YELLOW LED referred to as “STATUS”. This is a general purpose LED that by default simply flashes an acknowledgement when the “COMM TEST” button is pressed.

The host application may temporarily override the function of these LEDs when it may need to display its own error code or status. By convention, it is best that a host app just uses the yellow STATUS for an error code, but all are available. When “count” below is 0, it returns the LED back to its default behavior.

{"req":"card.aux"
”mode”:”monitor”
“name”:”yellow” or ”red” or ”green” to temporarily override the behavior of an LED
”count”:numberOfPulses the LED will display a repeating pattern of this many pulses, generally indicating a documented error code
”usb”:true sets LED behavior so that it is only displayed when on USB power, else it is off. This ensures that when a device is temporarily unplugged and on battery power that a perpetually-pulsing LED will not drain the battery.
} → {}

16 Request Protocol Implementation Considerations

Commands may be issued to the Notecard via USB, UART, or I2C. The Notecard UART interface is fixed at 9600 N/8/1.

The Notecard I2C Slave address, 0x17, is reconfigurable with a card.io request sent on USB or UART. The I2C interface implements a straightforward "serial over I2C" protocol. To ensure robustness and equitable bus sharing, I2C transactions should be sent at most every 1ms.

```
SendToNotecard(buffer, datalen):
    I2CWriteTransaction:
        Write(&datalen, 1)          # where datalen is between 1 and 255
        Write(buffer[:datalen])

data, datalen, available = ReceiveFromNotecard(datalen):
    I2CWriteTransaction:
        bytebuf = 0
        Write(&bytebuf, 1)          # indicates that a read is coming
        Write(&datalen, 1)          # where datalen is between 1 and 255
    I2CReadTransaction:
        Read(&readbuf, datalen+2)   # a 2-byte header is always returned
        available = readbuf[0]      # min num bytes still avail AFTER the read
        datalen = readbuf[1]        # num of valid data bytes returned in buffer
        data = readbuf[2:2+datalen]
```

17 Notehub Requests

Many of the requests above will work for Notehub in addition to Notecard. However, there are also certain requests that are available strictly on the Notehub.

17.1 **hub.env.get** can be used to read environment variables

{"req":"hub.env.get"

"mode":

"app" use the variables scoped at the Project-level

"fleet" use the variables scoped at the Fleet-level

"device" use the variables scoped at the Device-level

"name":"key" is the optional key name of the variable to be retrieved. If this is not specified, all keys and their values are retrieved

} → {

"text":"value" is the single value if a named key is specified on the request

"env":{"key":"value","key2","value2",…} if no named key is specified

}

17.2 **hub.env.set** can be used to set or delete environment variables

{"req":"hub.env.set"

"mode":

"app" use the variables scoped at the Project-level

"fleet" use the variables scoped at the Fleet-level

"device" use the variables scoped at the Device-level

"name":"key" is the key name of the variable to be set

"text":"value" is the optional value to be set. If not specified, the key is deleted.

} → {}

2018-12-05	Firmware	Feature complete
2019-08-22		Added AUX “track” mode to enable in-motion event counting/tracking
2020-02-03		Removed ‘personal’ projects as they were deprecated by user feedback
2020-03-12		Added card.motion
2020-03-23	9232	Added voltage-variable periods and mode string
2020-03-25	9235	Added “deleted” to note.get and note.changes
2020-03-31	9281	Added circular GPS ringfence
2020-04-06	9302	Added name/vvalue to card.voltage to make it easy for the host mcu to do voltage-variable timing behaviors just like the notecard itself
2020-04-06	9308	Remove manual SIM switching
2020-04-07		Simplify for the time being: - service.set “align” and “retry” now undocumented - service.set “minimum” now undocumented - card.attn “usb” is now undocumented - service.set “voff”, “vmin”, “align” all now undocumented - card.usage.rate is now undocumented - removed TLS info from service.set for custom notehub host - removed doc of ability to disable/enable comms ports
2020-04-13	9360	Made “file” an optional parameter on note.add and note.get, defaulting to “data.qo” and “data.qi” respectively.
2020-04-13	9360	Added “sync” parameter to service.set “continuous” so that the behavior of dynamically listening for service-modified files is an optional behavior – to optimize current using E-DRX on modem.
2020-04-14	9364	Added “alert” to service.log which flags high urgency
2020-04-21		Fixed doc of file.changes to have { rather than [for “info”
2020-04-21		Added doc about unanticipated fields in responses
2020-04-21		Added some notes about our use of JSON
2020-04-26	9525	Added card.wireless, removed wireless from card.io and card.version, added undocumented “radar” service.set mode
2020-04-28	9575	change return args on card.location so that status is returned in Status field just like card.location.mode
2020-04-28	9576	Remove redundant return args from card.location.mode
2020-05-01	9595	Reverted card.attn and documented seconds==0 behavior
2020-05-02	9605	Allow dev to track in notefile other than _track.qo
2020-05-07	9662	Deprecate note.template; now “template”:true arg on note.add
2020-05-08	9671	Eliminate confusion by clarifying “total” vs “changes” on the return arguments from note.changes and file.changes
2020-05-10	9703	After clear dev confusion, split out card.location.mode from card.location.track (while maintaining backward compatibility)
2020-05-11		Re-added service.set “minimum” and corrected doc of “off”

2020-05-14		Added hub.env.get and hub.env.set
2020-05-16		Clarified in card.motion that it status is a comma-separated list
2020-05-17	9900	Added motion requests
2020-05-18	9925	Added voltage-variable behavior to DFU via dfu.status
2020-05-19		Documented card.attn watchdog
2020-05-20		Greg: renamed env.time/env.location to card.*
2020-05-20		Clarified wording of card.time as it relates to zone
2020-05-22	9981	Changed default variable voltage to be “”
2020-05-23	10006	Changed ‘reset’ to ‘arm’ on card.attn, and added ‘disarm’
2020-05-25	10009	Changed semantic meaning of ‘seconds’ on card.attn
2020-06-02		Typo in card.motion.sync
2020-06-04	10255	Resurrect note.template and leave note.add template:true undocumented, so that it is clear that it isn’t adding a note
2020-06-10	10538	Add “minutes” to card.location.mode for ringfence debounce
2020-06-16	10657	Add new arguments to service.sync.status
2020-06-17	10670	Major change to change “service” to “hub” in requests
2020-06-23	10736	Added location ATTN trigger, and added more programmatically-testable state information to card.location’s status return
2020-06-30		Removed (unimplemented) payload from web transactions
2020-06-30	10777	Added file.changes.pending
2020-07-01		Fixed doc of card.motion.mode
2020-07-12	10818	Changed env.get to return a proper JSON array, not a text buf
2020-07-12	10831	Added env.default and env.set
2020-07-14	10881	Removed card.aux “button mode” because it has proven to be of no value to customers
2020-07-14	10881	Added card.aux “monitor mode” for devices that need a set of front-panel status indicators for testing/viewing comm status
2020-07-22	10961	Added 11, 12, 13, 14, 18, 12.1, 14.1, 18.1 note.template formats
2020-08-11	11113	Fixed the fact that YELLOW and GREEN were swapped in aux monitor mode