habitat

Release 0.2.0

November 21, 2014

Contents

Contents:

CHAPTER 1

Introduction

habitat is a system for uploading, processing, storing and displaying telemetry and related information transmitted from high altitude balloons.

Typically this telemetry takes the form of a GPS position and potentially other data, accompanied by information on who received the data, and is displayed by means of a map or chart (or both) showing the path a balloon took and the historic trend of sensor readings.

Internally, configuration and data is stored in a CouchDB database. The back end is written in Python and is responsible for parsing incoming data and storing it in the database, while the frontend is written independently in JavaScript and HTML and communicates with CouchDB directly to obtain data and display it.

This documentation covers setting up a habitat system, describes the format used to store data in CouchDB, and provides reference documentation for the habitat source code.

Useful habitat links:

- habitat on github
- habitat's continuous integration server
- this documentation
- habitat home page

CHAPTER 2

Installing

habitat is currently still in early development and is not ready to be installed.

Check back here later for installation help when the first release is out.

CHAPTER 3

Configuration

3.1 Command Line Configuration

habitat daemons takes zero or one command line arguments: an optional filename specifying the configuration to use, or "./habitat.yml" by default:

```
./bin/parser
# or
./bin/parser /path/to/config.yml
```

3.2 Configuration File

The configuration file is written in YAML as several key: value pairs. Various habitat components may require certain pieces of configuration; where possible these are all documented here.

3.2.1 habitat-wide Configuration

```
couch_uri: "http://localhost:5984"
couch_db: habitat
log_stderr_level: DEBUG
log_file_level: INFO
```

couch_uri and *couch_db* specify how to connect to the CouchDB database. The URI may contain authentication details if required.

log_stderr_level and *log_file_level* set the log levels for a log file and the stderr output and may be "NONE", "ERROR", "WARN", "INFO" or "DEBUG".

3.2.2 parser configuration

```
parser:
    certs_dir: "/path/to/certs"
    modules:
        - name: "UKHAS"
          class: "habitat.parser_modules.ukhas_parser.UKHASParser"
parserdaemon:
    log_file: "/path/to/parser/log"
```

Inside the *parser* and *parserdaemon* objects:

- *certs_dir* specifies where the habitat certificates (used for code signing) are kept
- *log_file* specifies where the parser daemon should write its log file to
- *modules* gives a list of all the parser modules that should be loaded, with a name (that must match names used in flight documents) and the Python path to load.

This configuration is used by *habitat.parser* and *habitat.parser_daemon*.

3.2.3 loadable_manager configuration

```
loadables:
    - name: "sensors.base"
      class: "habitat.sensors.base"
    - name: "sensors.stdtelem"
      class: "habitat.sensors.stdtelem"
    - name: "filters.common"
      class: "habitat.filters"
```

Inside the *loadables* object is a list of modules to load and the short name they should be loaded against. This is used by *habitat.loadable_manager*.

CHAPTER 4

Database information

4.1 Schema

habitat stores information in a CouchDB database. At present five types of document are stored, identified by a `type` key:

- Flight documents detailing a balloon flight (`type: "flight"`)
- Payload Configuration documents containing settings for one payload, such as radio transmission data and format (`type: "payload_configuration"`).
- Payload Telemetry documents containing parsed information from a telemetry message transmitted by a payload and associated with a Flight (`type: "payload_telemetry"`)
- Listener telemetry documents containing position data on someone listening to a payload (`type: "listener_telemetry"`)
- Listener information documents containing metadata on a listener such as name and radio (`type: "listener_information"`)

The schema are described using JSON Schema and the latest version may be browsed online via jsonschema explorer.

Database documents are typically managed through the various web interfaces and are uploaded and retrieved using the CouchDB API.

4.2 Views, Filters & Validation Functions

Documents in the habitat CouchDB are indexed and accessed using CouchDB views, which are pre-calculated sets of results that are updated automatically and may be paged and searched through.

A selection of generic views are provided, but it's entirely likely that a custom view would be required for a given application.

Three types of function may be defined in a CouchDB design document. Views consist of a map and optionally a reduce and are typically used to query stored data. Filters selectively include certain documents in a stream from the database, for example to the parser. Validation functions check all new incoming documents to ensure they meet whatever requirements are imposed, making sure that only valid documents are stored in the database.

For more comprehensive information, please refer to the CouchDB documentation.

4.3 Included Views

For documentation on the views currently included with habitat, please refer to the source documentation for each: *habitat.views*.

4.4 Using Views: Example

4.4.1 Simple Example

In this simple example we just fetch the list of *payload_configuration* documents and print out payload names. The full configuration document is available as *payload["doc"]*.

Python

```
import couchdbkit
db = couchdbkit.Server("http://habitat.habhub.org")["habitat"]
payloads = db.view("payload_configuration/name_time_created", include_docs=True)
for payload in payloads:
    print "Payload: {0}".format(payload["key"][0])
```

Javascript

```
<script src="jquery.couch.js"></script>

db = $.couch.db("habitat")
db.view("payload_configuration/name_time_created", {include_docs: true, success: function(data) {
    for(payload in data.rows) {
        console.log("Payload: " + data.rows[payload].key[0]);
    }
}});
```

4.4.2 View Collation

A more complicated example, which demonstrates view collation. The *flight/launch_time_including_payloads* view returns both *flight* documents and the associated *payload_configuration* documents, indicated by the second item in the key (0 for *flight*, 1 for *payload_configuration*). This code snippet fetches all the flights and prints their name and launch time and the name of each payload in them.

Python

```
import couchdbkit
db = couchdbkit.Server("http://habitat.habhub.org")["habitat"]
flights = db.view("flight/launch_time_including_payloads", include_docs=True)
for flight in flights:
    if flight["key"][1] == 0:
        print "Flight {0} launches at {1}!".format(
            flight["doc"]["name"], flight["doc"]["launch"]["time"])
        print "  Payloads:"
    else:
        print "     {0}".format(flight["doc"]["name"])
```

CHAPTER 5

Filters

5.1 Filter Levels

There are three points in the message flow at which a filter can act: before pre-parsing happens, after the pre-parse but before the main parse, and after the main parse.

Pre-parsing extracts a callsign from a string and uses it to look up the rest of the payload's configuration, so pre-parse filters are specified per module and will act on everything that module receives. These are called pre-filters and are specified in the parser configuration document.

Intermediate filters act after a callsign has been found but before the message is parsed for data, so they can correct particular format errors a certain payload might be transmitting. Post-parse filters act after the data parsing has happened, so can tweak the output data. Both intermediate and post filters are specified in the payload section of a flight document.

5.2 Filter Syntax

Two types of filters are supported: `normal` and `hotfix`. Normal filters give a callable object, which must be in the Python path, and optionally may give a configuration object which will be passed as the second argument to the callable. Hotfix filters just supply some Python code, which is used as the body of a function given the incoming data as its sole argument. In either case, the filter must return the newly processed data.

Example of a normal filter:

```
{
    "type": "normal",
    "callable": "habitat.filters.upper_case"
}

# habitat/filters.py
def upper_case(data):
    return data.upper()
```

A normal filter with a configuration object:

```
{
    "type": "normal",
    "callable": "habitat.filters.daylight_savings",
    "config": {"time_field": 7}
}
```

```
# habitat/filters.py
def daylight_savings(data, config):
    time = data[config['time_field']]
    hour = int(time[0:2])
    data[config['time_field']] = str(hour + 1) + time[2:]
    return data
```

A hotfix filter:

```
{
    "type": "hotfix",
    "code": "parts = data.split(',')\nreturn '.'.join(parts)\n"
}
```

Which would be assembled into:

```
def f(data):
    parts = data.split(',')
    return '.'.join(parts)
```

A more complete hotfix example, to fix non-zero-padded time values:

```
from habitat.utils.filtertools import UKHASChecksumFixer

parts = data.split(",")
timestr = parts[2]
timeparts = timestr.split(":")
timestr = "{0:02d}:{1:02d}:{2:02d}".format(*[int(part) for part in timeparts])
parts[2] = timestr
newdata = ",".join(parts)

with UKHASChecksumFixer('xor', {"data": data}) as fixer:
    fixer["data"] = newdata

    return fixer["data"]
```

5.3 Filter Utils

Please refer to *habitat.utils.filtertools* for information on available filter tools such as UKHASChecksumFixer used above.

CHAPTER 6

Certificates

The certificates folder contains x509 certificate files which are used to verify the authenticity of hotfix code.

As hotfix code is run live from the CouchDB database, habitat uses certificates to check that the code can be trusted. The habitat developers maintain a certificate authority, whose certificate is included as *ca/habitat_ca_cert.pem*, which is used to sign code-signing certificates.

Hotfix code then has its SHA-256 digest signed by the developer's private key, and this is verified by habitat before the code is executed.

You can add new CA certificates to the *ca* folder, and new code-signing certificates to the *certs* folder, as you please. Hotfix code references a certificate filename found in the *certs* folder.

6.1 Generating a Private Key

```
$ openssl genrsa -des3 4096 > private.pem
$ openssl req -new -key private.pem -out req.csr
```

Now send req.csr to us and we can sign it with the habitat CA and give you the signed certificate.

6.2 Generating a Certificate Authority

This is a fair bit more complex. Consider using tools such as tinyca or gnomint.

6.3 Signing Code

```
$ vi my_hotfix_code.py
$ habitat/bin/sign_hotfix my_hotfix_code.py ~/my_rsa_key.pem
```

The printed result is a JSON object which can be placed into the filters list on a flight document.

CHAPTER 7

UKHAS Parser Configuration

7.1 Introduction

The UKHAS protocol is the most widely used at time of writing, and is implemented by the UKHAS parser module. This document provides information on how what configuration settings the UKHAS parser module expects.

Parser module configuration is given in the “sentence” dictionary of the payload dictionary in a flight document.

7.2 Generating Payload Configuration Documents

The easiest and recommended way to generate configuration documents is using the web tool genpayload.

7.3 Standard UKHAS Sentences

A typical minimum UKHAS protocol sentence may be:

```
$$habitat,123,13:16:24,51.123,0.123,11000*ABCD
```

This sentence starts with a double dollar sign (`$$`) followed by the payload name (here `habitat`), several comma-delimited fields and is then terminated by an asterisk and four-digit CRC16 CCITT checksum (`*ABCD`).

In this typical case, the fields are a message ID, the time, a GPS latitude and longitude in decimal degrees, and the current altitude.

However, both the checksum algorithm in use and the number, type and order of fields may be configured per-payload.

7.4 Parser Module Configuration

The parser module expects to be given the callsign, the checksum algorithm, the protocol name (“UKHAS”) and a list of fields, each of which should at least specify the field name and data type.

7.4.1 Checksum Algorithms

Three algorithms are available:

- CRC16 CCITT (`crc16-ccitt`):

  The recommended algorithm, uses two bytes transmitted as four ASCII digits in hexadecimal. Can often be calculated using libraries for your payload hardware platform. In particular, note that we use a polynomial of 0x1021 and a start value of 0xFFFF, without reversing the input. If implemented correctly, the string `habitat` should checksum to 0x3EFB.

- XOR (`xor`):

  The simplest algorithm, calculating the one-byte XOR over all the message data and transmitting as two ASCII digits in hexadecimal. `habitat` checksums to 0x63.

- Fletcher-16 (`fletcher-16`):

  Not recommended but supported. Uses a modulus of 255 by default, if modulus 256 is required use `fletcher-16-256`.

In all cases, the checksum is of everything after the `$$` and before the `*`.

7.4.2 Field Names

Field names may be any string that does not start with an underscore. It is recommended that they follow the basic pattern of `prefix[_suffix[_suffix[...]]]` to aid presentation: for example, `temperature_internal` and `temperature_external` could then be grouped together automatically by a user interface.

In addition, several common field names have been standardised on, and their use is strongly encouraged:

Field	Name To Use	Notes
Sentence ID (aka count, message count, sequence number)	`sentence_id`	An increasing integer
Time	`time`	Something like HH:MM:SS or HHMMSS or HHMM or HH:MM.
Latitude	`latitude`	Will be converted to decimal degrees based on *format* field.
Longitude	`longitude`	Will be converted to decimal degrees based on *format* field.
Altitude	`altitude`	In, or converted to, metres.
Temperature	`temperature`	Should specify a suffix, such as `_internal` or `_external`. In or converted to degrees Celsius.
Satellites In View	`satellites`	
Battery Voltage	`battery`	Suffixes allowable, e.g., `_backup`, `_cutdown`, but without the suffix it is treated as the main battery voltage. In volts.
Pressure	`pressure`	Suffixes allowable, e.g., `_balloon`. Should be in or converted to Pa.
Speed	`speed`	For speed over the ground. Should be converted to m/s (SI units).
Ascent Rate	`ascentrate`	For vertical speed. Should be m/s.

Standard user interfaces will use title case to render these names, so `flight_mode` would become `Flight Mode` and so on. Some exceptions may be made in the case of the common field names specified above.

7.4.3 Field Types

Supported types are:

- `string`: a plain text string which is not interpreted in any way.
- `float`: a value that should be interpreted as a floating point number. Transmitted as a string, e.g., “123.45”, rather than in binary.

- `int`: a value that should be interpreted as an integer.
- `time`: a field containing the time as either `HH:MM:SS` or just `HH:MM`. Will be interpreted into a time representation.
- `time`: a field containing the time of day, in one of the following formats: `HH:MM:SS`, `HHMMSS`, `HH:MM`, `HHMM`.
- `coordinate`: a coordinate, see below

7.4.4 Coordinate Fields

Coordinate fields are used to contain, for instance, payload latitude and longitude. They have an additional configuration parameter, `format`, which is used to define how the coordinate should be parsed. Options are:

- `dd.dddd`: decimal degrees, with any number of digits after the decimal point. Leading zeros are allowed.
- `ddmm.mm`: degrees and decimal minutes, with the two digits just before the decimal point representing the number of minutes and all digits before those two representing the number of degrees.

In both cases, the number can be prefixed by a space or + or - sign.

Please note that the the options reflect the style of coordinate (degrees only vs degrees and minutes), not the number of digits in either case.

7.4.5 Units

Received data may use any convenient unit, however it is strongly recommended that filters (see below) be used to convert the incoming data into SI units. These then allow for standardisation and ease of display on user interface layers.

7.4.6 Filters

See *Filters*

CHAPTER 8

habitat code documentation

habitat	The top level habitat package.

8.1 habitat

The top level habitat package.

habitat is an application for tracking the flight path of high altitude balloons, relying on a network of users with radios sending in received telemetry strings which are parsed into position information and displayed on maps.

See http://habitat.habhub.org for more information.

habitat.parser	Interpret incoming telemetry strings into useful telemetry data.
habitat.parser_daemon	Run the Parser as a daemon connected to CouchDB's _changes feed.
habitat.parser_modules	Parser modules for specific protocols.
habitat.loadable_manager	Load configured Python functions for later use elsewhere.
habitat.sensors	Sensor function libraries.
habitat.filters	Commonly required filters that are supplied with habitat.
habitat.uploader	Python interface to document insertion into CouchDB.
habitat.utils	Various utilities for general use by `habitat`.
habitat.views	View functions for CouchDB with the couch-named-python view server, used by habitat relatec

8.1.1 habitat.parser

Interpret incoming telemetry strings into useful telemetry data.

Classes

Parser(config)	habitat's parser
ParserModule(parser)	Base class for real ParserModules to inherit from.

class `habitat.parser.`**`Parser`**(*config*)

habitat's parser

`Parser` takes arbitrary unparsed payload telemetry and attempts to use each loaded `ParserModule` to turn this telemetry into useful data.

On construction, it will:

- Use `config[daemon_name]` as `self.config` (defaults to 'parser').
- Load modules from `self.config["modules"]`.
- Connects to CouchDB using `self.config["couch_uri"]` and `config["couch_db"]`.

parse(*doc*, *initial_config=None*)
Attempts to parse telemetry information out of a new telemetry document *doc*.

This function attempts to determine which of the loaded parser modules should be used to parse the message, and which payload_configuration document it should be given to do so (if *initial_config* is specified, no attempt will be made to find any other configuration document).

The resulting parsed document is returned, or None is returned if no data could be parsed.

Some field names in data["data"] are reserved, as indicated by a leading underscore.

These fields may include:

- `_protocol` which gives the parser module name that was used to decode this message

From the UKHAS parser module in particular:

- `_sentence` gives the ASCII sentence from the UKHAS parser

Parser modules should be wary when outputting field names with leading underscores.

class `habitat.parser.`**`ParserModule`**(*parser*)
Base class for real ParserModules to inherit from.

ParserModules are classes which turn radio strings into useful data. They do not have to inherit from `ParserModule`, but can if they want. They must implement `pre_parse()` and `parse()` as described below.

pre_parse(*string*)
Go though *string* and attempt to extract a callsign, returning it as a string. If *string* is not parseable by this module, raise `CantParse`. If *string* might be parseable but no callsign could be extracted, raise `CantExtractCallsign`.

parse(*string*, *config*)
Go through *string* which has been identified as the format this parser module should be able to parse, extracting the data as per the information in *config*, which is the `sentence` dictionary extracted from the payload's configuration document.

8.1.2 habitat.parser_daemon

Run the Parser as a daemon connected to CouchDB's _changes feed.

Classes

`ParserDaemon`(config[, daemon_name])	`ParserDaemon` runs persistently, watching CouchDB's _changes feed

class `habitat.parser_daemon.`**`ParserDaemon`**(*config*, *daemon_name='parserdaemon'*)
`ParserDaemon` runs persistently, watching CouchDB's _changes feed for new unparsed telemetry, parsing it with `Parser` and storing the result back in the database.

On construction, it will:

- Connect to CouchDB using `self.config["couch_uri"]` and `config["couch_db"]`.

run()
Start a continuous connection to CouchDB's _changes feed, watching for new unparsed telemetry.

8.1.3 habitat.parser_modules

Parser modules for specific protocols.

`habitat.parser_modules.ukhas_parser`	This module contains the parser for the UKHAS telemetry protoco
`habitat.parser_modules.simple_binary_parser`	This module contains a parser for a generic and simple binary prot

habitat.parser_modules.ukhas_parser

This module contains the parser for the UKHAS telemetry protocol format.

The protocol is most succinctly described as:

```
$$<callsign>,<data>,<data>,...,<data>*<checksum>
```

The typical minimum telemetry string is:

```
$$<callsign>,<message number>,<time>,<latitude>,<longitude>,<altitude>,<data>,...,<data>*<checksum>
```

The number of custom data fields and their types are configurable.

Data fields are typically human readable (or at the least ASCII) readings of sensors or other system information. See the `habitat.sensors` module for more information on supported formats.

Checksums work on the message content between the `$$` and the `*`, non-inclusive, and are given as hexadecimal (upper or lower case) after the `*` in the message.

Supported checksums are CRC16-CCITT with polynomial 0x1021 and start 0xFFFF, Fletcher-16 and an 8bit XOR over the characters. The corresponding values for configuration are `crc16-ccitt`, `fletcher-16` and `xor`. For compatibility, a varient of Fletcher16 using modulus 256 is also provided, as `fletcher-16-256`. Don't use it for new payloads. `none` may also be specified as a checksum type if no checksum is used; in this case the message should not include a terminating `*`.

See also:

UKHAS Parser Configuration

Classes

`ParserModule`(parser)	Base class for real ParserModules to inherit from.
`UKHASParser`(parser)	The UKHAS Parser Module

class `habitat.parser_modules.ukhas_parser.`**UKHASParser**(*parser*)
The UKHAS Parser Module

pre_parse(*string*)
Check if *string* is parsable by this module.

If it is, `pre_parse()` extracts the payload name and return it. Otherwise, a `ValueError` is raised.

parse(*string*, *config*)

Parse *string*, extracting processed field data.

config is a dictionary containing the sentence dictionary from the payload's configuration document.

Returns a dictionary of the parsed data, with field names as keys and the result as the value. Also inserts a `payload` field containing the payload name, and an `_sentence` field containing the ASCII sentence that data was parsed from.

`ValueError` is raised on invalid messages.

habitat.parser_modules.simple_binary_parser

This module contains a parser for a generic and simple binary protocol. The protocol does not specify callsigns or checksums, so the assumption is that both are provided in an outer protocol layer or out of band. Any binary data that python's struct.unpack may interpret is usable.

Any fields may be submitted but it is recommended that a GPS-provided latitude, longitude and time are submitted.

The configuration document should specify the format string, a name and optionally a sensor for each field in the data. The format strings will be concatenated to unpack the data. A format string prefix may be provided as the `format_prefix` key in the configuration, to specify byte order, size and alignment. Note that at present each field must map to precisely one format string argument, so while variable length strings are OK, a field cannot have, for instance, two integers.

Example `payload_configuration.sentences[0]`:

```
{
    "protocol": "simple_binary",
    "callsign": "1234567890",
    "format_prefix": "<",
    "fields": [
        {
            "format": "i",
            "name": "latitude"
        }, {
            "format": "i",
            "name": "longitude"
        }, {
            "format": "I",
            "name": "date",
            "sensor": "std_telem.binary_timestamp"
        }, {
            "format": "b",
            "name": "temperature"
        }
    ],
    "filters": {
        "post": [
            {
                "type": "normal",
                "filter": "common.numeric_scale",
                "source": "latitude",
                "scale": 1E-7
            },
            {
                "type": "normal",
                "filter": "common.numeric_scale",
                "source": "longitude",
                "scale": 1E-7
            }
        ]
    }
}
```

For the list of format string specifiers, please see: http://docs.python.org/2/library/struct.html.

The filter `common.numeric_scale` may be useful for fixed-point data rather than sending floats, and the various `std_telem.binary*` sensors may be applicable.

Classes

`ParserModule`(parser)	Base class for real ParserModules to inherit from.
`SimpleBinaryParser`(parser)	The Simple Binary Parser Module

Exceptions

`CantExtractCallsign`	Parser submodule cannot find a callsign, though in theory might be able to parse the sentence if one we

class `habitat.parser_modules.simple_binary_parser.`**`SimpleBinaryParser`**(*parser*)

The Simple Binary Parser Module

`pre_parse`(*string*)

As no callsign is provided by the protocol, assume any string we are given is potentially parseable binary data.

`parse`(*data*, *config*)

Parse *string*, extracting processed field data.

config is the relevant sentence dictionary from the payload's configuration document, containing the required binary format and field details.

Returns a dictionary of the parsed data.

ValueError is raised on invalid messages.

8.1.4 habitat.loadable_manager

Load configured Python functions for later use elsewhere.

The manager is configured with modules to use and it loads all the functions defined in each module's `__all__`. This ensures that users cannot specify arbitrary paths in runtime configuration which may lead to undesired or insecure behaviour.

The modules that functions are loaded from are given shorthand names to ease referring to them elsewhere, for example `habitat.sensors.stdtelem.time()` might become `stdtelem.time`. The shorthand is specified in the configuration document.

Configuration

loadable_manager reads its configuration data from the config argument to *LoadableManager.__init__*, which is typically parsed from the configuration YAML file in the following format:

```
loadables:
    - name: "sensors.base"
      class: "habitat.sensors.base"
    - name: "filters.common"
      class: "habitat.filters"
```

`name` specifies the shorthand name that the module will be available under; it should begin either `sensors` or `filters` for use by the respective parts of habitat, which prepend the relevant prefix themselves.

For example, to use the filter `habitat.filters.semicolons_to_commas()` in a flight document, having configured as above, you would specify:

```
"filters": {
    "intermediate": {
        [
            {
                "type": "normal",
                "filter": "common.semicolons_to_commas"
            }
        ]
    }
}
```

Sensor Functions

One of the major uses of *loadable_manager* (and historically its only use) is sensor functions, used by parser modules to convert input data into usable Python data formats. See `habitat.sensors` for some sensors included with habitat, but you may also want to write your own for a specific type of data.

A sensor function may two one or two arguments, *config* and *data* or just *data*. It can return any Python object which can be stored in the CouchDB database.

config is a dict of options. It is passed to the function from `LoadableManager.run()`

data is the string to parse.

Filter Functions

Another use for the *loadable_manager* is filters that are applied against incoming telemetry strings. Which filters to use is specified in a payload's flight document, either as user-specified (but signed) hotfix code or a loadable function name, as with sensors.

See `habitat.filters` for some filters included with habitat.

Filters can take one or two arguments, *config*, *data* or just *data*. They should return a suitably modified form of data, optionally using anything from *config* which was specified by the user in the flight document.

Classes

`LoadableManager`(config)	The main Loadable Manager class.

class `habitat.loadable_manager.`**`LoadableManager`**(*config*)

The main Loadable Manager class.

On construction, all modules listed in config["loadables"] will be loaded using `load()`.

load (*module*, *shorthand*)
Loads *module* as a library and assigns it to *shorthand*.

run (*name*, *config*, *data*)
Run the loadable specified by *name*, giving it *config* and *data*.

If the loadable only takes one argument, it will only be given *data*. *config* is ignored in this case.

Returns the result of running the loadable.

8.1.5 habitat.sensors

Sensor function libraries.

`habitat.sensors.base`	Basic sensor functions.
`habitat.sensors.stdtelem`	Sensor functions for dealing with telemetry.

habitat.sensors.base

Basic sensor functions.

These sensors cover simple ASCII representations of numbers and strings.

Functions

`ascii_float`(config, data)	Parse *data* to a float.
`ascii_int`(config, data)	Parse *data* to an integer.
`constant`(config, data)	Checks that *data* is equal to config[”expect”], returning None.
`string`(data)	Returns *data* as a string.

`habitat.sensors.base.`**ascii_int** (*config*, *data*)
Parse *data* to an integer.

`habitat.sensors.base.`**ascii_float** (*config*, *data*)
Parse *data* to a float.

`habitat.sensors.base.`**string** (*data*)
Returns *data* as a string.

`habitat.sensors.base.`**constant** (*config*, *data*)
Checks that *data* is equal to config[”expect”], returning None.

`habitat.sensors.base.`**binary_b64** (*data*)
Encodes raw binary data to base64.

habitat.sensors.stdtelem

Sensor functions for dealing with telemetry.

Functions

`coordinate`(config, data)	Parses ASCII latitude or longitude into a decimal-degrees float.
`strptime`((string, format) -> struct_time)	Parse a string to a time tuple according to a format specification.
`time`(data)	Parse the time, validating it and returning the standard `HH:MM:SS`.

`habitat.sensors.stdtelem.`**`time`**`(`*data*`)`

Parse the time, validating it and returning the standard `HH:MM:SS`.

Accepted formats include `HH:MM:SS`, `HHMMSS`, `HH:MM` and `HHMM`. It uses strptime to ensure the values are sane.

`habitat.sensors.stdtelem.`**`coordinate`**`(`*config*, *data*`)`

Parses ASCII latitude or longitude into a decimal-degrees float.

Either decimal degrees or degrees with decimal minutes are accepted (degrees, minutes and seconds are not currently supported).

The format is specified in `config["format"]` and can look like either `dd.dddd` or `ddmm.mmmm`, with one to three leading `d` characters and one to six trailing `d` or `m` characters.

`habitat.sensors.stdtelem.`**`binary_timestamp`**`(`*data*`)`

Parse a four byte unsigned integer into a time string in the format “HH:MM:SS”. Date information is thus discarded.

`habitat.sensors.stdtelem.`**`binary_bcd_time`**`(`*data*`)`

Parse two or three bytes (given as a string, format `2s` or `3s`) into hours, minutes and optionally seconds in the format “HH:MM:SS”.

8.1.6 habitat.filters

Commonly required filters that are supplied with habitat.

Filters are small functions that can be run against incoming payload telemetry during the parse phase, either before attempts at callsign extraction, before the actual parse (but after the callsign has been identified) or after parsing is complete.

This module contains commonly used filters which are supplied with habitat, but end users are free to write their own and have `habitat.loadable_manager` load them for use.

Functions

`invalid_always`(data)	Add the _fix_invalid key to data.
`invalid_gps_lock`(config, data)	Checks a gps_lock field to see if the payload has a lock
`invalid_location_zero`(data)	If the latitude and longitude are zero, the fix is marked invalid.
`numeric_scale`(config, data)	Post filter that scales a key from *data* by a factor in *config*.
`semicolons_to_commas`(config, data)	Intermediate filter that converts semicolons to commas.
`simple_map`(config, data)	Post filter that maps source to destination values based on a dictionary.

`habitat.filters.`**`semicolons_to_commas`**`(`*config*, *data*`)`

Intermediate filter that converts semicolons to commas.

All semicolons in the string are replaced with colons and the checksum is updated; `crc16-ccitt` is assumed by default but can be overwritten with `config["checksum"]`.

```
>>> semicolons_to_commas({}, '$$testpayload,1,2,3;4;5;6*8A24')
'$$testpayload,1,2,3,4,5,6*888F'
```

`habitat.filters.`**`numeric_scale`**`(`*config*, *data*`)`

Post filter that scales a key from *data* by a factor in *config*.

`data[config["source"]]` is multiplied by `config["factor"]` and written back to `data[config["destination"]]` if it exists, or `data[config["source"]]` if not. `config["offset"]` is also optionally applied along with `config["round"]`.

```
>>> config = {"source": "key", "factor": 2.0}
>>> data = {"key": "4", "other": "data"}
>>> numeric_scale(config, data) == {'key': 8.0, 'other': 'data'}
True
>>> config["destination"] = "result"
>>> numeric_scale(config, data) == {'key': 8.0, 'result': 16.0, 'other':
...     'data'}
...
True
```

`habitat.filters.`**`simple_map`**`(`*config*, *data*`)`

Post filter that maps source to destination values based on a dictionary.

`data[config["source"]]` is used as a lookup key in `config["map"]` and the resulting value is written to `data[config["destination"]]` if it exists, or `data[config["source"]]` if not.

A `ValueError` is raised if `config["map"]` is not a dictionary or does not contain the value read from *data*. >>> config = {“source”: “key”, “destination”: “result”, “map”: ... {1: ‘a’, 2: ‘b’}} ... >>> data = {“key”: 2} >>> simple_map(config, data) == {‘key’: 2, ‘result’: ‘b’} True

`habitat.filters.`**`invalid_always`**`(`*data*`)`

Add the _fix_invalid key to data.

`habitat.filters.`**`invalid_location_zero`**`(`*data*`)`

If the latitude and longitude are zero, the fix is marked invalid.

`habitat.filters.`**`invalid_gps_lock`**`(`*config*, *data*`)`

Checks a gps_lock field to see if the payload has a lock

The source key is config[”source”], or “gps_lock” if that is not set.

The fix is marked invalid if data[source] is not in the list config[”ok”].

`habitat.filters.`**`zero_pad_coordinates`**`(`*config*, *data*`)`

Post filter that inserts zeros after the decimal point in coordinates, to fix the common error of having the integer and fractional parts of a decimal degree value as two ints and outputting them using something like *sprintf(“%i.%i”, int_part, frac_part);*, resulting in values that should be 51.0002 being output as 51.2 or similar.

The fields to change is the list *config[”fields”]* and the correct post-decimal-point width is *config[”width”]*. By default fields is *[”latitude”, “longitude”]* and width is 5.

`habitat.filters.`**`zero_pad_times`**`(`*config*, *data*`)`

Intermediate filter that zero pads times which have been incorrectly transmitted as e.g. *12:3:8* instead of *12:03:08*. Only works when colons are used as delimiters.

The field position to change is *config[”field”]* and defaults to 2 (which is typical with $$PAYLOAD,ID,TIME). The checksum in use is *config[”checksum”]* and defaults to *crc16-ccitt*.

8.1.7 habitat.uploader

Python interface to document insertion into CouchDB.

The uploader is a client for end users that pushes documents into a CouchDB database where they can be used directly by the web client or picked up by a daemon for further processing.

Classes

`Extractor`()	A base class for an Extractor.
`ExtractorManager`(uploader)	Manage one or more `Extractor` objects, and handle their logging.
`UKHASExtractor`()	
`Uploader`(callsign[, couch_uri, couch_db, ...])	An easy interface to insert documents into a habitat CouchDB.
`UploaderThread`()	An easy wrapper around `Uploader` to make a non blocking Uploader

Exceptions

`CollisionError`	
`UnmergeableError`	Couldn't merge a `payload_telemetry` CouchDB conflict after many tries.

exception `habitat.uploader.`**`UnmergeableError`**

Couldn't merge a `payload_telemetry` CouchDB conflict after many tries.

class `habitat.uploader.`**`Uploader`**(*callsign*, *couch_uri='http://habitat.habhub.org/'*, *couch_db='habitat'*, *max_merge_attempts=20*)

An easy interface to insert documents into a habitat CouchDB.

This class is intended for use by a listener.

After having created an `Uploader` object, call `payload_telemetry()`, `listener_telemetry()` or `listener_information()` in any order. It is however recommended that `listener_information()` and `listener_telemetry()` are called once before any other uploads.

`flights()` returns a list of current flight documents.

Each method that causes an upload accepts an optional kwarg, time_created, which should be the unix timestamp of when the doc was created, if it is different from the default 'now'. It will add time_uploaded, and turn both times into RFC3339 strings using the local offset.

See the CouchDB schema for more information, both on validation/restrictions and data formats.

`listener_telemetry`(*data*, *time_created=None*)

Upload a `listener_telemetry` doc. The `doc_id` is returned

A `listener_telemetry` doc contains information about the listener's current location, be it a rough stationary location or a constant feed of GPS points. In the former case, you may only need to call this function once, at startup. In the latter, you might want to call it constantly.

The format of the document produced is described elsewhere; the actual document will be constructed by `Uploader`. *data* must be a dict and should typically look something like this:

```
data = {
    "time": "12:40:12",
    "latitude": -35.11,
    "longitude": 137.567,
    "altitude": 12
}
```

`time` is the GPS time for this point, `latitude` and `longitude` are in decimal degrees, and `altitude` is in metres.

`latitude` and `longitude` are mandatory.

Validation will be performed by the CouchDB server. *data* must not contain the key `callsign` as that is added by `Uploader`.

`listener_information`(*data*, *time_created=None*)

Upload a listener_information doc. The doc_id is returned

A listener_information document contains static human readable information about a listener.

The format of the document produced is described elsewhere (TODO?); the actual document will be constructed by `Uploader`. *data* must be a dict and should typically look something like this:

```
data = {
    "name": "Adam Greig",
    "location": "Cambridge, UK",
    "radio": "ICOM IC-7000",
    "antenna": "9el 434MHz Yagi"
}
```

data must not contain the key `callsign` as that is added by `Uploader`.

`payload_telemetry`(*string*, *metadata=None*, *time_created=None*)

Create or add to the `payload_telemetry` document for *string*.

This function attempts to create a new `payload_telemetry` document for the provided string (a new document, with one receiver: you). If the document already exists in the database it instead downloads it, adds you to the list of receivers, and reuploads.

metadata can contain extra information about your receipt of *string*. Nothing has been standardised yet (TODO), but here's an example of what you might be able to do in the future:

```
metadata = {
    "frequency": 434075000,
    "signal_strength": 5
}
```

metadata must not contain the keys `time_created`, `time_uploaded`, `latest_listener_information` or `latest_listener_telemetry`. These are added by `Uploader`.

`flights`()

Return a list of flight documents.

Finished flights are not included; so the returned list contains active and not yet started flights (i.e., now <= flight.end).

Only approved flights are included.

Flights are sorted by end time.

Active is (flight.start <= now <= flight.end), i.e., within the launch window.

The key `_payload_docs` is added to each flight document and is populated with the documents listed in the payloads array, provided they exist. If they don't, that _id will be skipped.

payloads()
Returns a list of all payload_configuration docs ever.

Sorted by name, then time created.

class habitat.uploader.**UploaderThread**
An easy wrapper around Uploader to make a non blocking Uploader

After creating an UploaderThread object, call start() to create a thread. Then, call settings() to initialise the underlying Uploader. You may then call any of the 4 action methods from Uploader with exactly the same arguments. Note however, that they do not return anything (see below for flights() returning).

Several methods may be overridden in the UploaderThread. They are:

- log()
- warning()
- saved_id()
- initialised()
- reset_done()
- caught_exception()
- got_flights()
- got_payloads()

Please note that these must all be thread safe.

If initialisation fails (bad arguments or similar), a warning will be emitted but the UploaderThread will continue to exist. Further calls will just emit warnings and do nothing until a successful settings() call is made.

The reset() method destroys the underlying Uploader. Calls will emit warnings in the same fashion as a failed initialisation.

start()
Start the background UploaderThread

join()
Asks the background thread to exit, and then blocks until it has

settings(**args*, ***kwargs*)
See Uploader's initialiser

reset()
Destroys the Uploader object, disabling uploads.

payload_telemetry(**args*, ***kwargs*)
See Uploader.payload_telemetry()

listener_telemetry(**args*, ***kwargs*)
See Uploader.listener_telemetry()

listener_information(**args*, ***kwargs*)
See Uploader.listener_information()

flights()
See Uploader.flights().

Flight data is passed to got_flights().

payloads()
See `Uploader.payloads()`.

Flight data is passed to `got_payloads()`.

debug(*msg*)
Log a debug message

log(*msg*)
Log a generic string message

warning(*msg*)
Alike log, but more important

saved_id(*doc_type*, *doc_id*)
Called when a document is succesfully saved to couch

initialised()
Called immiediately after successful Uploader initialisation

reset_done()
Called immediately after resetting the Uploader object

caught_exception()
Called when the Uploader throws an exception

got_flights(*flights*)
Called after a successful flights download, with the data.

Downloads are initiated by calling `flights()`

got_payloads(*payloads*)
Called after a successful payloads download, with the data.

Downloads are initiated by calling `payloads()`

class habitat.uploader.**ExtractorManager**(*uploader*)
Manage one or more `Extractor` objects, and handle their logging.

The extractor manager maintains a list of `Extractor` objects. Any `push()` or `skipped()` calls are passed directly to each added Extractor in turn. If any Extractor produces logging output, or parsed data, it is returned to the `status()` and `data()` methods, which the user should override.

The ExtractorManager also handles thread safety for all Extractors (i.e., it holds a lock while pushing data to each extractor). Your `status()` and `data()` methods should be thread safe if you want to call the Extractor-Manager from more than one thread.

uploader: an `Uploader` or `UploaderThread` object

add(*extractor*)
Add the extractor object to the manager

push(*b*, ***kwargs*)
Push a received byte of data, b, to all extractors.

b must be of type str (i.e., ascii, not unicode) and of length 1.

Any kwargs are passed to extractors. The only useful kwarg at the moment is the boolean "baudot hack".

baudot_hack is set to True when decoding baudot, which doesn't support the '*' character, as the UKHA-SExtractor needs to know to replace all '#' characters with '*'s.

skipped(*n*)
Tell all extractors that approximately n undecodable bytes have passed

This advises extractors that some bytes couldn't be decoded for whatever reason, but were transmitted. This can assist some fixed-size packet formats in recovering from errors if one byte is dropped, say, due to the start bit being flipped. It also causes Extractors to 'give up' after a certain amount of time has passed.

status (*msg*)
Logging method, called by Extractors when something happens

data (*d*)
Called by Extractors if they are able to parse extracted data

class `habitat.uploader.`**Extractor**
A base class for an Extractor.

An extractor is responsible for identifying telemetry in a stream of bytes, and extracting them as standalone strings. This may be by using start/end delimiters, or packet lengths, or whatever. Extracted strings are passed to `Uploader.payload_telemetry()` via the ExtractorManager.

An extractor may optionally attempt to parse the data it has extracted. This does not affect the upload of extracted data, and offical parsing is done by the habitat server, but may be useful to display in a GUI. It could even be a stripped down parser capable of only a subset of the full protocol, or able to parse the bare minimum only. If it succeeds, the result is passed to `ExtractorManager.data()`.

push (*b*, ***kwargs*)
see `ExtractorManager.push()`

skipped (*n*)
see `ExtractorManager.skipped()`

8.1.8 habitat.utils

Various utilities for general use by `habitat`.

`habitat.utils.checksums`	Various checksum calculation utilities.
`habitat.utils.dynamicloader`	A generic dynamic python module loader.
`habitat.utils.filtertools`	Various utilities for filters to call upon.
`habitat.utils.startup`	Useful functions for daemon startup
`habitat.utils.immortal_changes`	An extension to couchdbkit's changes consumer that never dies.
`habitat.utils.quick_traceback`	Quick traceback module shortcuts for logging

habitat.utils.checksums

Various checksum calculation utilities.

Functions

`crc16_ccitt`(data)	Calculate the CRC16 CCITT checksum of *data*.
`fletcher_16`(data[, modulus])	Calculate the Fletcher-16 checksum of *data*, default modulus 255.
`op_xor`	xor(a, b) – Same as a ^ b.
`xor`(data)	Calculate the XOR checksum of *data*.

`habitat.utils.checksums.`**crc16_ccitt** (*data*)
Calculate the CRC16 CCITT checksum of *data*.

(CRC16 CCITT: start 0xFFFF, poly 0x1021)

Returns an upper case, zero-filled hex string with no prefix such as `0A1B`.

```
>>> crc16_ccitt("hello,world")
'E408'
```

`habitat.utils.checksums.`**`xor`**`(`*data*`)`

Calculate the XOR checksum of *data*.

Returns an upper case, zero-filled hex string with no prefix such as `01`.

```
>>> xor("hello,world")
'2C'
```

`habitat.utils.checksums.`**`fletcher_16`**`(`*data*, *modulus=255*`)`

Calculate the Fletcher-16 checksum of *data*, default modulus 255.

Returns an upper case, zero-filled hex string with no prefix such as `0A1B`.

```
>>> fletcher_16("hello,world")
'6C62'
>>> fletcher_16("hello,world", 256)
'6848'
```

habitat.utils.dynamicloader

A generic dynamic python module loader.

The main function to call is load(). In addition, several functions to quickly test the loaded object for certain conditions are provided:

- `isclass()`
- `isfunction()`
- `isgeneratorfunction()`
- `isstandardfunction()` (`isfunction and not isgeneratorfunction`)
- `iscallable()`
- `issubclass()`
- `hasnumargs()`
- `hasmethod()`
- `hasattr()`

Further to that, functions `expectisclass()`, `expectisfunction()`, etc., are provided which are identical to the above except they raise either a ValueError or a TypeError where the original function would have returned `False`.

Example use:

```
def loadsomething(loadable):
    loadable = dynamicloader.load(loadable)
    expectisstandardfunction(loadable)
    expecthasattr(loadable, 2)
```

If you use `expectiscallable()` note that you may get either a function or a class, an object of which is callable (i.e., the class has `__call__(self, ...)`). In that case you may need to create an object:

```
if isclass(loadable):
    loadable = loadable()
```

Of course if you've used `expectisclass()` then you will be creating an object anyway. Note that classes are technically "callable" in that calling them creates objects. `expectiscallable()` ignores this.

A lot of the provided tests are imported straight from inspect and are therefore not documented here. The ones implemented as a part of this module are.

Functions

`expecthasattr`(*args, **kwargs)	does *thing* have an attribute named *attr*?
`expecthasmethod`(*args, **kwargs)	is *loadable.name* callable?
`expecthasnumargs`(*args, **kwargs)	does *thing* have *num* arguments?
`expectiscallable`(*args, **kwargs)	is *loadable* a method, function or callable class?
`expectisclass`(*args, **kwargs)	is *thing* a class?
`expectisfunction`(*args, **kwargs)	is *thing* a function? (either normal or generator)
`expectisgeneratorfunction`(*args, **kwargs)	is *thing* a generator function?
`expectisstandardfunction`(*args, **kwargs)	is *thing* a normal function (i.e., not a generator)
`expectissubclass`(*args, **kwargs)	is *thing* a subclass of *the other thing*?
`fullname`(loadable)	Determines the full name in `module.module.class` form
`hasattr`(thing, attr)	does *thing* have an attribute named *attr*?
`hasmethod`(loadable, name)	is *loadable.name* callable?
`hasnumargs`(thing, num)	does *thing* have *num* arguments?
`iscallable`(loadable)	is *loadable* a method, function or callable class?
`isclass`(thing)	is *thing* a class?
`isfunction`(thing)	is *thing* a function? (either normal or generator)
`isgeneratorfunction`(thing)	is *thing* a generator function?
`isstandardfunction`(thing)	is *thing* a normal function (i.e., not a generator)
`issubclass`(thing, the_other_thing)	is *thing* a subclass of *the other thing*?
`load`(loadable[, force_reload])	Attempts to dynamically load *loadable*

`habitat.utils.dynamicloader.`**`load`**`(`*loadable*, *force_reload=False*`)`

Attempts to dynamically load *loadable*

loadable: a class, a function, a module, or a string that is a dotted-path to one a class function or module

Some examples:

```
load(MyClass) # returns MyClass
load(MyFunction) # returns MyFunction
load("mypackage") # returns the mypackage module
load("packagea.packageb") # returns the packageb module
load("packagea.packageb.aclass") # returns aclass
```

`habitat.utils.dynamicloader.`**`fullname`**`(`*loadable*`)`

Determines the full name in `module.module.class` form

loadable: a class, module or function.

If fullname is given a string it will `load()` it in order to resolve it to its true full name.

`habitat.utils.dynamicloader.`**`isclass`**`(`*thing*`)`

is *thing* a class?

`habitat.utils.dynamicloader.`**`isfunction`**`(`*thing*`)`

is *thing* a function? (either normal or generator)

`habitat.utils.dynamicloader.`**`isgeneratorfunction`**`(`*thing*`)`

is *thing* a generator function?

`habitat.utils.dynamicloader.`**`issubclass`**`(`*thing*, *the_other_thing*`)`
    is *thing* a subclass of *the other thing*?

`habitat.utils.dynamicloader.`**`hasattr`**`(`*thing*, *attr*`)`
    does *thing* have an attribute named *attr*?

`habitat.utils.dynamicloader.`**`isstandardfunction`**`(`*thing*`)`
    is *thing* a normal function (i.e., not a generator)

`habitat.utils.dynamicloader.`**`hasnumargs`**`(`*thing*, *num*`)`
    does *thing* have *num* arguments?

If *thing* is a function, the positional arguments are simply counted up. If *thing* is a method, the positional arguments are counted up and one is subtracted in order to account for `method(self, ...)` If *thing* is a class, the positional arguments of `cls.__call__` are counted up and one is subtracted (self), giving the number of arguments a callable object created from that class would have.

`habitat.utils.dynamicloader.`**`hasmethod`**`(`*loadable*, *name*`)`
    is *loadable.name* callable?

`habitat.utils.dynamicloader.`**`iscallable`**`(`*loadable*`)`
    is *loadable* a method, function or callable class?

For *loadable* to be a callable class, an object created from it must be callable (i.e., it has a `__call__` method)

`habitat.utils.dynamicloader.`**`expectisclass`**`(`**args*, ***kwargs*`)`
    is *thing* a class?

`habitat.utils.dynamicloader.`**`expectisfunction`**`(`**args*, ***kwargs*`)`
    is *thing* a function? (either normal or generator)

`habitat.utils.dynamicloader.`**`expectisgeneratorfunction`**`(`**args*, ***kwargs*`)`
    is *thing* a generator function?

`habitat.utils.dynamicloader.`**`expectisstandardfunction`**`(`**args*, ***kwargs*`)`
    is *thing* a normal function (i.e., not a generator)

`habitat.utils.dynamicloader.`**`expectiscallable`**`(`**args*, ***kwargs*`)`
    is *loadable* a method, function or callable class?

For *loadable* to be a callable class, an object created from it must be callable (i.e., it has a `__call__` method)

`habitat.utils.dynamicloader.`**`expectissubclass`**`(`**args*, ***kwargs*`)`
    is *thing* a subclass of *the other thing*?

`habitat.utils.dynamicloader.`**`expecthasnumargs`**`(`**args*, ***kwargs*`)`
    does *thing* have *num* arguments?

If *thing* is a function, the positional arguments are simply counted up. If *thing* is a method, the positional arguments are counted up and one is subtracted in order to account for `method(self, ...)` If *thing* is a class, the positional arguments of `cls.__call__` are counted up and one is subtracted (self), giving the number of arguments a callable object created from that class would have.

`habitat.utils.dynamicloader.`**`expecthasmethod`**`(`**args*, ***kwargs*`)`
    is *loadable.name* callable?

`habitat.utils.dynamicloader.`**`expecthasattr`**`(`**args*, ***kwargs*`)`
    does *thing* have an attribute named *attr*?

habitat.utils.filtertools

Various utilities for filters to call upon.

Classes

UKHASChecksumFixer(protocol, data)	A utility to help filters modify data that has been checksummed.

class habitat.utils.filtertools.**UKHASChecksumFixer**(*protocol*, *data*)

A utility to help filters modify data that has been checksummed. It may be used as a context manager or via a class method.

For use as a context manager:

Specify the protocol in use with *protocol* and pass in the string being modified as `data["data"]`, then use the return value as a dictionary whose `data` key you can modify as you desire. On exit, the checksum of that string is then updated if the original string's checksum was valid.

If the original checksum was invalid, the original string is output instead.

```
>>> data = {"data": "$$hello,world*E408"}
>>> with UKHASChecksumFixer('crc16-ccitt', data) as fixer:
...     fixer["data"] = "$$hi,there,world*E408"
...
>>> fixer["data"]
'$$hi,there,world*39D3'
```

For direct calling as a class method:

Call UKHASChecksumFixer.fix(protocol, old_data, new_data). The function will either return new_data with a fixed checksum if the original checksum was valid, or it will return the old_data if the original checksum was invalid.

```
>>> UKHASChecksumFixer.fix('crc16-ccitt',
...                        "$$hello,world*E408", "$$hi,there,world*E408")
'$$hi,there,world*39D3'
```

habitat.utils.startup

Useful functions for daemon startup

Functions

load_config()	Loads the habitat config.
main(main_class)	Main function for habitat daemons.
setup_logging(config, daemon_name)	**setup_logging** initalises the Python logging module.

Classes

null_logger([level])	A python logging handler that discards log messages silently.

habitat.utils.startup.**load_config**()

Loads the habitat config.

The path to the configuration YAML file can be specified as the single command line argument (read from `sys.argv[1]`) or will default to `./habitat.yml`.

class `habitat.utils.startup.`**`null_logger`**`(`*level=0*`)`

A python logging handler that discards log messages silently.

Initializes the instance - basically setting the formatter to None and the filter list to empty.

`habitat.utils.startup.`**`setup_logging`**`(`*config*, *daemon_name*`)`

setup_logging initalises the `Python logging module`.

It will initalise the 'habitat' logger and creates one, two, or no Handlers, depending on the values provided for `log_file_level` and `log_stderr_level` in *config*.

`habitat.utils.startup.`**`main`**`(`*main_class*`)`

Main function for habitat daemons. Loads config, sets up logging, and runs.

`main_class.__name__.lower()` will be used as the config sub section and passed as *daemon_name*.

main_class specifies a class from which an object will be created. It will be initialised with arguments (config, daemon_name) and then the method run() of the object will be invoked.

habitat.utils.immortal_changes

An extension to couchdbkit's changes consumer that never dies.

Classes

`Consumer`	Mock out external modules that might annoy documentation build systems.

habitat.utils.quick_traceback

Quick traceback module shortcuts for logging

Functions

`oneline`([exc_value])	Return a single line describing 'exc_value'

`habitat.utils.quick_traceback.`**`oneline`**`(`*exc_value=None*`)`

Return a single line describing 'exc_value'

exc_value shold be either an Exception instance, for example, acquired via 'except ValueError as e:'; or None, in which case the exception currently being handled is used.

The string returned is the last line of Python's normal traceback; something like 'ValueError: some message', with no newline.

8.1.9 habitat.views

View functions for CouchDB with the couch-named-python view server, used by habitat related design documents.

`habitat.views.flight`	Functions for the flight design document.
`habitat.views.listener_information`	Functions for the listener_information design document.
`habitat.views.listener_telemetry`	Functions for the listener_telemetry design document.

Continued on next page

Table 8.24 – continued from previous page

habitat.views.payload_telemetry	Functions for the payload_telemetry design document.
habitat.views.payload_configuration	Functions for the payload_configuration design document.
habitat.views.habitat	Functions for the core habitat design document.
habitat.views.parser	Functions for the parser design document.
habitat.views.utils	Shared utility functions for views.

habitat.views.flight

Functions for the flight design document.

Contains schema validation and views by flight launch time, window end time and payload name and window end time.

Functions

all_name_map(doc)	View: flight/all_name
end_start_including_payloads_map(doc)	View: flight/end_start_including_payloads
launch_time_including_payloads_map(doc)	View: flight/launch_time_including_payloads
read_json_schema(schemaname)	
rfc3339_to_timestamp	Mock out external modules that might annoy documentation build systems
validate(new, old, userctx, secobj)	Validate this flight document against the schema, then check that only man
validate_doc(data, schema)	Validate *data* against *schema*, raising descriptive errors
version	Mock out external modules that might annoy documentation build systems

Exceptions

ForbiddenError	Mock out external modules that might annoy documentation build systems.
UnauthorizedError	Mock out external modules that might annoy documentation build systems.

habitat.views.flight.**end_start_including_payloads_map**(*doc*)

View: flight/end_start_including_payloads

Emits:

```
[end_time, start_time, flight_id, 0] -> [payload_configuration ids]
[end_time, start_time, flight_id, 1] -> {linked payload_configuration doc 1}
[end_time, start_time, flight_id, 1] -> {linked payload_configuration doc 2}
...
```

Or, when a flight has no payloads:

```
[end_time, start_time, flight_id, 0] -> null
```

Times are all UNIX timestamps (and therefore in UTC).

Sorts by flight window end time then start time.

If the flight has payloads, emit it with the list of payloads, and emit a link for each payload so that they get included with include_docs. If a flight does not have payloads, it is emitted by itself.

Only shows approved flights.

Used by the parser to find active flights and get the configurations used to decode telemetry from them.

May otherwise be used to find upcoming flights and their associated payloads, though typically the view `launch_time_including_payloads` would be more useful as it sorts by launch time.

Query using `startkey=[current_timestamp]` to get all flights whose windows have not yet ended. Use `include_docs=true` to have the linked payload_configuration documents fetched and returned as the `"doc"` key for that row, otherwise the row's value will just contain an object that holds the linked ID. See the CouchDB documentation for details on linked documents.

`habitat.views.flight.`**`launch_time_including_payloads_map`**`(`*doc*`)`

View: `flight/launch_time_including_payloads`

Emits:

```
[launch_time, flight_id, 0] -> [payload_configuration ids]
[launch_time, flight_id, 1] -> {linked payload_configuration doc 1}
[launch_time, flight_id, 1] -> {linked payload_configuration doc 2}
...
```

Or, when a flight has no payloads:

```
[launch_time, flight_id, 0] -> null
```

Times are all UNIX timestamps (and therefore in UTC).

Sort by flight launch time.

Only shows approved flights.

Used by the calendar and other interface elements to show a list of upcoming flights.

Query using `startkey=[current_timestamp]` to get all upcoming flights. Use `include_docs=true` to have the linked payload_configuration documents fetched and returned as the `"doc"` key for that row, otherwise the row's value will just contain an object that holds the linked ID. See the CouchDB documentation for details on linked documents.

`habitat.views.flight.`**`unapproved_name_including_payloads_map`**`(`*doc*`)`

View: `flight/unapproved_name_including_payloads`

Emits:

```
[name, flight_id, 0] -> [payload_configuration ids]
[name, flight_id, 1] -> {linked payload_configuration doc 1}
[name, flight_id, 1] -> {linked payload_configuration doc 2}
...
```

Or, when a flight has no payloads:

```
[name, flight_id, 0] -> null
```

Times are all UNIX timestamps (and therefore in UTC).

Sort by flight name.

Only shows unapproved flights.

Used by the administration approval interface to list unapproved flights.

Use `include_docs=true` to have the linked payload_configuration documents fetched and returned as the `"doc"` key for that row, otherwise the row's value will just contain an object that holds the linked ID. See the CouchDB documentation for details on linked documents.

`habitat.views.flight.`**`all_name_map`**`(`*doc*`)`

View: `flight/all_name`

Emits:

```
[name] -> null
```

Sort by flight name.

Show all flights, even those unapproved.

Used where the UI must show all the flights in some usefully searchable sense, for instance when creating a new flight document based on some old or unapproved one, or when approving new flight documents.

habitat.views.listener_information

Functions for the listener_information design document.

Contains schema validation and a view by creation time and callsign.

Functions

`callsign_time_created_map`(doc)	View: `listener_information/callsign_time_created`
`must_be_admin`(user[, msg])	Raise UnauthorizedError if the user is not an admin
`read_json_schema`(schemaname)	
`rfc3339_to_timestamp`	Mock out external modules that might annoy documentation build systems.
`time_created_callsign_map`(doc)	View: `listener_information/time_created_callsign`
`validate`(new, old, userctx, secobj)	Only allow admins to edit/delete and validate the document against the schema for listen
`validate_doc`(data, schema)	Validate *data* against *schema*, raising descriptive errors
`version`	Mock out external modules that might annoy documentation build systems.

`habitat.views.listener_information.`**`time_created_callsign_map`**`(`*doc*`)`

View: `listener_information/time_created_callsign`

Emits:

```
[time_created, callsign] -> null
```

Times are UNIX timestamps (and therefore in UTC).

Sorts by time created. Useful to see the latest listener information.

`habitat.views.listener_information.`**`callsign_time_created_map`**`(`*doc*`)`

View: `listener_information/callsign_time_created`

Emits:

```
[callsign, time_created] -> null
```

Times are UNIX timestamps (and therefore in UTC).

Sorts by callsign. Useful to see a certain callsign's latest information.

habitat.views.listener_telemetry

Functions for the listener_telemetry design document.

Contains schema validation and a view by creation time and callsign.

Functions

callsign_time_created_map(doc)	View: listener_telemetry/callsign_time_created
must_be_admin(user[, msg])	Raise UnauthorizedError if the user is not an admin
read_json_schema(schemaname)	
rfc3339_to_timestamp	Mock out external modules that might annoy documentation build systems.
time_created_callsign_map(doc)	View: listener_telemetry/time_created_callsign
validate(new, old, userctx, secobj)	Only allow admins to edit/delete and validate the document against the schema for listen
validate_doc(data, schema)	Validate *data* against *schema*, raising descriptive errors
version	Mock out external modules that might annoy documentation build systems.

habitat.views.listener_telemetry.**time_created_callsign_map**(*doc*)

View: `listener_telemetry/time_created_callsign`

Emits:

```
[time_created, callsign] -> null
```

Times are UNIX timestamps (and therefore in UTC).

Sorts by time created. Useful to see the latest listener telemetry.

habitat.views.listener_telemetry.**callsign_time_created_map**(*doc*)

View: `listener_telemetry/callsign_time_created`

Emits:

```
[callsign, time_created] -> null
```

Times are UNIX timestamps (and therefore in UTC).

Sorts by callsign. Useful to see a certain callsign's latest telemetry.

habitat.views.payload_telemetry

Functions for the payload_telemetry design document.

Contains schema validation and a view by flight, payload and received time.

Functions

flight_payload_time_map(doc)	View: payload_telemetry/flight_payload_time
payload_time_map(doc)	View: payload_telemetry/payload_time
read_json_schema(schemaname)	
rfc3339_to_timestamp	Mock out external modules that might annoy documentation build systems.
validate(new, old, userctx, secobj)	Validate this payload_telemetry document against the schema, then perform
validate_doc(data, schema)	Validate *data* against *schema*, raising descriptive errors
version	Mock out external modules that might annoy documentation build systems.

Exceptions

ForbiddenError	Mock out external modules that might annoy documentation build systems.
UnauthorizedError	Mock out external modules that might annoy documentation build systems.

habitat.views.payload_telemetry.**flight_payload_time_map**(*doc*)

View: `payload_telemetry/flight_payload_time`

Emits:

```
[flight_id, payload_configuration_id, estimated_time_received] -> null
```

Useful to find telemetry related to a certain flight.

habitat.views.payload_telemetry.**payload_time_map**(*doc*)

View: `payload_telemetry/payload_time`

Emits:

```
[payload_configuration_id, estimated_time_received] -> null
```

Useful to find telemetry related to a specific payload_configuration.

habitat.views.payload_telemetry.**time_map**(*doc*)

View: `payload_telemetry/time`

Emits:

```
estimated_time_received -> is_flight_telemetry
```

Useful to get recent telemetry uploaded to habitat.

This can also be used to make a simple map application. It's worth noting that such a technique is a bit of a bodge, since estimated_time_received will not necessarily (but could) update if another receiver is added to the doc, so asking this view for all telemetry since min(the last poll, the most recent telemetry I have) is not infallible. That said, doing a proper sync is quite difficult.

habitat.views.payload_telemetry.**add_listener_update**(*doc*, *req*)

Update function: `payload_telemetry/_update/add_listener`

Given a prototype payload_telemetry JSON document in the request body, containing just the _raw telemetry string and one entry in receivers, create the document or merge this listener into it as appropriate.

Used by listeners when a new payload telemetry string has been received.

Usage:

```
PUT /habitat/_design/payload_telemetry/_update/add_listener/<doc ID>

{
    "data": {
        "_raw": "<base64 raw telemetry data>"
    },
    "receivers": {
        "<receiver callsign>": {
            "time_created": "<RFC3339 timestamp>",
            "time_uploaded": "<RFC3339 timestamp>",
            <other keys as desired, for instance
             latest_listener_telemetry, latest_listener_info, etc>
        }
    }
}
```

The document ID should be sha256(doc["data"]["_raw"]) in hexadecimal.

Returns "OK" if everything was fine, otherwise CouchDB will raise an error. Errors might occur in validation (in which case the validation error is returned) or because of a save conflict. In the event of a save conflict, uploaders should retry the same request until the conflict is resolved.

`habitat.views.payload_telemetry.`**`http_post_update`**`(`*doc*, *req*`)`

Update function: `payload_telemetry/_update/http_post`

Creates a new payload_telemetry document with all keys present in the HTTP POST form data available in `doc.data._fallbacks` and the `from` HTTP querystring key as the receiver callsign if available. The `data` field will be base64 encoded and used as `doc.data._raw`.

This function has additional functionality specific to RockBLOCKs: if all of the keys `imei`, `momsn`, `transmit_time`, `iridium_latitude`, `iridium_longitude`, `iridium_cep` and `data` are present in the form data, then: * `imei` will be copied to `doc.data._fallbacks.payload` so it can be

used as a payload callsign.

- `iridium_latitude` and `iridium_longitude` will be copied to `doc.data._fallbacks.latitude` and `longitude` respectively.
- `data` will be hex decoded before base64 encoding so it can be directly used by the binary parser module.
- `transmit_time` will be decoded into an RFC3339 timestamp and used for the `time_created` field in the receiver section.
- `transmit_time` will be decoded into hours, minutes and seconds and copied to `doc.data._fallbacks.time`.

Usage:

```
POST /habitat/_design/payload_telemetry/_update/http_post?from=callsign

data=hello&imei=whatever&so=forth
```

This update handler may not currently be used on existing documents or with a PUT request; such requests will fail.

Returns "OK" if everything was fine, otherwise CouchDB will return a (hopefully instructive) error.

habitat.views.payload_configuration

Functions for the payload_configuration design document.

Contains schema validation and a view by payload name and configuration version.

Functions

`callsign_time_created_index_map`(doc)	View: `payload_configuration/callsign_time_created_index`
`must_be_admin`(user[, msg])	Raise UnauthorizedError if the user is not an admin
`name_time_created_map`(doc)	View: `payload_configuration/name_time_created`
`only_validates`(doc_type)	
`read_json_schema`(schemaname)	
`rfc3339_to_timestamp`	Mock out external modules that might annoy documentation build systems.
`validate`(new, old, userctx, secobj)	Validate payload_configuration documents against the schema and then against
`validate_doc`(data, schema)	Validate *data* against *schema*, raising descriptive errors

`habitat.views.payload_configuration.`**`name_time_created_map`**`(`*doc*`)`

View: `payload_configuration/name_time_created`

Emits:

```
[name, time_created] -> null
```

In the key, `time_created` is emitted as a UNIX timestamp (seconds since epoch).

Used to get a list of all current payload configurations, for display purposes or elsewhere where sorting by name is useful.

`habitat.views.payload_configuration.`**`callsign_time_created_index_map`**`(`*doc*`)`

View: `payload_configuration/callsign_time_created_index`

Emits:

```
[callsign, time_created, 1] -> [metadata, sentence 1]
[callsign, time_created, 2] -> [metadata, sentence 2]
...
[callsign, time_created, n] -> [metadata, sentence n]
```

Where `metadata` is:

```
{
    "name": doc.name,
    "time_created": doc.time_created (original string),
    "metadata": doc.metadata (if present in doc)
}
```

(In other words, one row per sentence in this document).

In the key, `time_created` is emitted as a UNIX timestamp (seconds since epoch).

Useful to obtain configuration documents for a given callsign if it can't be found via upcoming flights, for example parsing test telemetry or selecting a sentence to copy when making a new document.

habitat.views.habitat

Functions for the core habitat design document.

Contains a validation function that applies to every document.

Functions

`must_be_admin`(user[, msg])	Raise UnauthorizedError if the user is not an admin
`validate`(new, old, userctx, secobj)	Core habitat validation function.
`version`	Mock out external modules that might annoy documentation build systems.

Exceptions

`ForbiddenError`	Mock out external modules that might annoy documentation build systems.

`habitat.views.habitat.`**`validate`**`(`*new*, *old*, *userctx*, *secobj*`)`

Core habitat validation function.

- Prevent deletion by anyone except administrators.
- Prevent documents without a type.
- Prevent documents whose type is invalid.

- Prevent changing document type.

habitat.views.parser

Functions for the parser design document.

Contains a filter to select unparsed payload_telemetry.

Functions

`unparsed_filter`(doc, req)	Filter: `parser/unparsed`
`version`	Mock out external modules that might annoy documentation build systems.

`habitat.views.parser.`**`unparsed_filter`**(*doc*, *req*)

Filter: `parser/unparsed`

Only select unparsed payload_telemetry documents.

habitat.views.utils

Shared utility functions for views.

Functions

`datetime_to_timestamp`	
`must_be_admin`(user[, msg])	Raise UnauthorizedError if the user is not an admin
`parse`	
`read_json_schema`(schemaname)	
`rfc3339_to_datetime`	
`rfc3339_to_timestamp`	
`rfc3339_to_utc_datetime`	
`timegm`	
`validate`	
`validate_doc`(data, schema)	Validate *data* against *schema*, raising descriptive errors
`validate_rfc3339`	Mock out external modules that might annoy documentation build systems.

Classes

`tzutc`

Exceptions

`ForbiddenError`	Mock out external modules that might annoy documentation build systems.
`UnauthorizedError`	Mock out external modules that might annoy documentation build systems.
`ValidationError`	

`habitat.views.utils.`**`must_be_admin`**`(`*user*, *msg='Only server administrators may edit this document.'*`)`
: Raise UnauthorizedError if the user is not an admin

`habitat.views.utils.`**`validate_doc`**`(`*data*, *schema*`)`
: Validate *data* against *schema*, raising descriptive errors

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*