Developing an Asystom Sentinel codec and testing it

Objective

This repo contains example source code to implement an Asystom Sentinel codec and a basic codec tester. This document provides some information on the source code and on a codec testing tool.

Available source code

Disclaimer

The source code available here is not intended for operational use. It is provided as a way to help understand the structure of Asystom Sentinel frames, and as an example of how to write an Asystom Sentinel codec.

Also, please note that this source code is not intended to build a LoRaWAN Codec API Specification-compatible codec. A specific Github repository (asystom-sentinel-codec) is dedicated to this goal. Still, this repository contains an example of codec that supports the API, and provides some functionalities that are not available in a LoRaWAN Codec API Specification-compatible codec due to limitations on the standard runtime environment of such a codec – i.e. an LNS runtime environment. This point is detailed below.

Source code files

The main files proposed here as examples to develop such a codec are :

- `sentinel_frame.js` – a class to represent a Sentinel frame and to provide related services, in particular decoding of the frame using the LoRaWAN Codec API Specification (as provided by functions in file `asystom_sentinel_codec.js`) or not
- `asystom_sentinel_codec.js` – a file containing functions compliant with the LoRaWAN Codec API Specification
- `frame_utilities.js` – a utility class to determine LoRa transmission data of a received frame such as the spreading factor, data rate and bandwidth
- `firmware_status.js` – utility classes to extract data from a Sentinel system status frame (the frame sent by a Sentinel device at startup time), such as device health information and firmware status
- `frame_factory.js` – a factory (in the GOF sense) to generate a Sentinel frame instance based on a LoRaWAN message as transmitted by an LNS. To do so, it uses services provided by LNS-specific classes found in files such as `kerlinkwmc_frame.js`, `multitech_frame.js`, `nifi_frame.js`, `loriot_frame.js`, `chirpstack_frame.js`, etc. These LNS-specific files handle the transformation of a raw frame (i.e. as provided by an LNS) into a canonical frame (i.e. a frame which can be processed generically by the Asystom NodeRED back-end)
- `physical_value.js` – a class to represent physical values as acquired by a Sentinel devices; it also provides symbols to represent various information about

the configuration of a Sentinel device (sensor identifiers, Sentinel vector data types, FFT zoom-related configuration information…)

- `segmented_frame.js` – a class to manage the aggregation of frame segments to produce a Sentinel frame.

In addition to these codec building blocks, the repository contains a higher-level file that allows to **test a codec** : `codec_tester.js`.

Sentinel frame decoder how-to

The source code provided here decodes an Asystom Sentinel frame Javascript object, an instance of `SentinelFrame` class, by calling its `decode()` method. The class instance constructor takes as input a JavaScript object that represents what is produced by an LNS communicating over HTTP.

As there is yet no LoRaWAN standard to describe how an LNS should organize the data it transmits to an application server, every LNS produces a different output.

Managing the various formats is performed by method `makeCanonical()` that actually produces a generic (canonical) representation of a Sentinel frame based on the LNS output. This method represents an interface to be implemented by a derived class of `SentinelFrame` class.

This repository contains a Sentinel frame factory (`FrameFactory` class) that supports several types of frame as produced by a LoRaWAN Network Server such as Loriot, Actility, Chirpstack and others. Derived classes for common LNSs are proposed : `LoriotFrame`, `ActilityFrame`, `ChirpstackFrame`, `MultitechFrame` and `NifiFrame`. The LoRaWAN message transmitted by the LNS is passed to static method `createFrame()` of this factory. This method guesses the type of LNS that produced the LoRaWAN message based on the message organisation and content then returns an instance of the right `SentinelFrame`-derived class.

The `decode()` method of this `SentinelFrame` instance can then be invoked to extract application data (battery level, temperature, signature data, FFT zoom data, device configuration information, ...) from the frame.

Codec tester

This program can be used to test an Asystom Sentinel decoder, be it the LoRaWAN Alliance Codec API Specification-compliant one or a full-fledged one. Once decoded, the frame content is printed on standard output.

The program supports frame decoding only.

The full-fledged decoder supports any type of frame :

- Data frame with only scalar values (battery voltage and temperature)
- Data frame with signature data
- Data frame with FFT zoom values
- A mix of the previous data frames (scalar values with either signature data or FFT zoom values)
- System status frame (a frame transmitted by a Sentinel device at boot time which describes the current device configuration in terms of RF parameters, vibration/sound analysis configuration, scheduling parameters for data uplink, etc.).

(See the Asystom Advisor and Asystom Sentinel documentation for details on uplinked information.)

The program fetches Asystom Sentinel frames from one or several data sources (NodeRED instances running the Asystom Advisor back-end) and decodes them.

This program supports four launch arguments :

- `--use-lorawan-codec` to force the use of the LoRaWAN codec, otherwise the full-fledged one is used
- `--request-frequency <frequency>` to specify the frequency at which the data sources must be polled. The frequency must be expressed in milliseconds.
- `--hide-result` to tell the program to not display the acquired physical values
- `--logging-on` to display log messages.

For instance, the following output is displayed if the LoRaWAN-compliant codec is used to decode a simple frame containing two scalar values.

```
{
  data: {
    scalarValues: [
      {
        name: "Ambient temperature",
        unit: "°C",
        value: 25.18467841611357,
        min: undefined,
        max: undefined,
      },
      {
        name: "Humidity",
        unit: "% rH",
        value: 12.860303654535745,
        min: undefined,
        max: undefined,
      },
    ],
  },
  errors: [
  ],
  warnings: [
  ],
}
```

The NodeRED servers that provide input frames must be specified in a JSON file called `uplink_sources.json`. It is an array of structures with the following format.

```
{
  "host": "<host domain URL>",
  "basic_auth": "<client HTTP basic authentication to the NodeRED
server>"
}
```

For instance, the following content represents a valid input file.

```
[
    {
        "host": "server.valid.domain.com",
        "basic_auth": "RGlkeW91cmVhbGx5ZGVjb2RldGhpc3N0cmluZz8="
    },
    {
        "host": "server.another.valid.domain.com",
        "basic_auth":
"Q29uZ3JhdHMsIHlvdSd2ZSBmb3VuZCBhbiBFYXN0ZXIgZWdnICE="
    },
    {
        "host": "server.yet.another.valid.domain.com",
        "basic_auth": "QXN5c3RvbSBTZW50aW5lbCBMb1JhV0FOIGNvZGVj"
    },
]
```

The following is an example of command to start the codec tester using the full-fledged codec, with logging activated and a refresh frequency of 1 minute.

```
node --logging-on --request-frequency 60000 codec_tester.js
```