WxflowToChords README

```
python3 WxflowToChords config.json
```

About

(See *Raspbian Installation* below for O/S installation instructions).

WxflowToChords are a set of python modules for converting Weatherflow json formatted datagrams into the CHORDS REST api, and submitting the data to a CHORDS portal.

There are four python modules:

- FromWxflow: Capture datagrams and put them in a queue. This is multithreaded, so that datagram reading can occur in parallel with other processing.
- DecodeWxflow: Translate the wxflow messages into structured data matching the CHORDS REST api. A JSON based configuration specifies the translation.
- ToChords: Send the structured data to a CHORDS portal. This is multithreaded, so that http writing can occur in parallel with other processing. The same JSON configuration provides other information for the CHORDS connection.
- WxflowToChords: Strings all three modules together for the end-to-end process.

Each module expects a configuration, provided in a JSON file. A single file can contain the configuration for all modules, or a file can be created for just the items needed for a gven module.

Each module will run a test case if it is invoked individually.

This code will be run on both regular computers, and micropython systems such as the Raspberry Pi Zero W. The code has been testyed against python3 on MacOS and Raspian Lite.

Weatherflow JSON Schema

Input data is structured according to the Weatherflow UDP JSON schema. Messages have a `type` field identifying the type of message, status fields indicating identity and hardware health, and in some cases an `obs` array containing the observed values, in a predefined order.

Unfortunately, the meaning of each `obs` array element is not identified in the message; you have to refer to the documentation to determine this. Other values, such as voltage and rssi, can be located by identifier.

For example:

```
{
  "serial_number":"HB-00000001",
  "type":"hub_status",
  "firmware_revision":"13",
  "uptime":1670133,
  "rssi":-62,
  "timestamp":1495724691,
  "reset_flags": 234881026,
  "stack": "1616,1608"
}
```

```
{
  "serial_number": "SK-00008453",
  "type":"rapid_wind",
  "device_id":1110,
  "hub_sn": "HB-00000001",
  "obs":[1493322445,2.3,128]
}
```

```
{
  "serial_number": "AR-00004049",
  "type": "device_status",
  "hub_sn": "HB-00000001",
  "timestamp": 1510855923,
  "uptime": 2189,
  "voltage": 3.50,
  "firmware_revision": 17,
  "rssi": -17,
  "sensor_status": 0
}
```

The WeatherFlow documentation seems to be in flux, so be sure to capture some datagrams to verify what they are transmitting.

FromWxflow

This module captures the broadcast datagrams from the weather station hub. It requires a JSON configuration file containing:

```
{
  "listen_port": 5022
}
```

DecodeWxflow

A JSON structure defines the mapping between the wxflow input data and the CHORDS portal api. A collection of wxflow decoders are defined (`wxflow_decoders`), for each message that is to be translated. Each one contains a list of match attributes. If an incoming messages matches one of the `wxflow_decoders`, that entry is used to decode the message.

The `wxflow_type` in `wxflow_decoders` (e.g. "ObsAir") are user-assigned, and have no special meaning.

Within each message decoding specification, there are two types of keys. If one begins with an underscore, it is a directive to the decoder, such as `_enabled` or `_match`. Otherwise, it is a key that will match a field in the incoming message, and the element directs further message handling.

The `obs` element has special meaning. It contains instructions on how to route the elements of an wxflow `obs` array.

If a CHORDS variable is identified as `at`, it will be converted to a timestamp and used for the `at=` timestamp.

```
{
  "chords_host": "chords_host.com",
  "listen_port": 50222,
```

```
    "skey": "123456",
    "wipy_report: {
      "_enabled": true,
      "_chords_inst_id" : "1",
      "_battv" : "wipyv"
    },
    "wxflow_decoders": [
      {
        "_enabled": true,
        "_wxflow_type": "ObsAir",
        "_chords_inst_id": "1",
        "_match": {
          "type": "obs_air",
          "serial_number": "AR-00005436"
        },
        "obs":[
            [0, "at"],
            [1, "pres"],
            [2, "tdry"],
            [3, "rh"],
            [4, "lcount"],
            [5, "ldist"],
            [6, "vbat"]
          ]
      },
      {
        "_enabled": true,
        "_wxflow_type": "HubStatus",
        "_chords_inst_id": "1",
        "_match": {
          "type": "hub_status",
          "serial_number": "HB-00004236"
        },
        "timestamp": {
          "chords_var": "at"
        },
        "rssi" :{
          "chords_var": "rssihub"
        }
      },
      {
        "_enabled": true,
        "_wxflow_type": "StationStatus",
        "_chords_inst_id": "1",
        "_match": {
          "type": "station_status",
          "serial_number": "AR-00005436"
        },
        "timestamp": {
          "chords_var": "at"
        },
        "voltage": {
            "chords_var": "vair"
        },
        "rssi": {
          "chords_var": "rssiair"
        },
        "sensor_status": {
          "chords_var": "statair"
        }
      }
    ]
  }
```

ToChords

This module takes the CHORDS data structure, reformats it as a URL, and sents it to a CHORDS instance as an http GET. There are two steps in this process, to bulid the URL and then submit it for transmission. This can be seen in `WxflowToChords`:

```
  uri = ToChords.buildURI(host, chords_record)
  ToChords.submitURI(uri)
```

`ToChords` requires a JSON configuration file containing the host name for the CHORDS instance, and the access key that is included in the GET.

```
{
  "chords_host": "chords_host.com",
  "skey": "key"
}
```

WxflowToChords

This module strings the three preceding ones together. It's the best place to see how the modules work.

Example processing, showing the wxflow datagram followed by the CHORDS structured data:

```
{"serial_number":"HB-
00004236","type":"hub_status","firmware_version":"26","uptime":88638,"rssi":-58,"timestamp":151145614
8,"reset_flags":503316482}
{
  "inst_id": "1",
  "skey": "123456",
  "vars": {
    "at": 1511456148,
    "rssihub": -58
  }
}
{"serial_number":"AR-00005436","type":"station_status","hub_sn":"HB-
00004236","timestamp":1511456154,"uptime":1825691,"voltage":3.46,"version":20,"rssi":-73,"sensor_stat
us":4}
{
  "inst_id": "1",
  "skey": "123456",
  "vars": {
    "at": 1511456154,
    "rssiair": -73,
    "statair": 4,
    "vair": 3.46
  }
}
{"serial_number":"AR-00005436","type":"obs_air","hub_sn":"HB-00004236","obs":
[[1511456154,770.00,13.43,33,0,0,3.46,1]],"firmware_revision":20}
{
  "inst_id": "1",
  "skey": "123456",
  "vars": {
    "at": 1511456154,
    "lcount": 0,
    "ldist": 0,
    "pres": 770.0,
    "rh": 33,
    "tdry": 13.43,
    "vbat": 3.46
  }
}
```

Raspbian Installation

1. Download raspbian. Buster was the current debian O/S; be sure to use this one or later.

2. Install balenaEtcher, and use it to flash the image to a 16GB micro-SD card.

3. ssh is not enabled by default. Enable it by mounting the flashed SD card, and add a file named *ssh* in the top directory. On MacOS, it's:

   ```
   touch /Volumes/boot/ssh
   ```

4. You can enable wifi connection an access point at initial boot, by creating a wifi configuration and placing it in *wpa_supplicant.conf*. You can set this up ahead of time; otherwise you can connect over ethernet and use *raspi-config* to automatically attach to the access point.

   To pre-configure wifi access to an existing access point, create *boot/wpa_supplicant.conf*:

   ```
   country=US
   ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
   update_config=1
   network={
     ssid="NETWORK-NAME"
     psk="NETWORK-PASSWORD"
   }
   ```

5. Power up the Pi *(with an Ethernet connection if wifi was not pre-configuerd)*, and ssh in:

   ```
   ssh pi@raspberrypi  # password raspberry
   ```

6. Configure the system name (and possibly the wifi network) using the configuration tool:

   ```
   sudo raspi-config
   ```

Micropython Notes (Deprecated)

I tried to get this working on a WiPy. It would run for indeterminate periods, and then throw OSError exceptions in the urequests code. After this, the networking would not work until after a reboot. So, I ripped out the WiPy specifics, andd transfered the project to a Raspberry Pi Zero W. Here are a few thoings learned while working with the WiPy.

General

Micropython is a slimmed down embedded version of python. The Ur-project seems to be micropython.org, but i have learned that many device developers (pycom, adafruit, microbit) have forked this project, and incompatibilities may exist. It seems that most of the customizations are related to the particular hardware set, but that doesn't mean that they will track the micropython.org releases.

On macOS

The goal is to have this running on a micropython embedded system. Fortunately, there is a micropython implmentation for Linux, macOS and Windows. This has been useful for testing the code apart from the embedded hardware. However, my experieince has been that the macOS version is not bug-for-bug identical to the micropython board that I have been using (the WiPy, and so this will only get you soo far.

The general idea for bringing up micropython on macOS:

```
brew install libffi
git clone --recurse https://github.com/micropython/micropython.git
cd micropython/ports/unix
make axtls
PKG_CONFIG_PATH=/usr/local/opt/libffi/lib/pkgconfig make install
./micropython
>>> import upip
>>> upip.install('micropython-socket')
>>> upip.install('micropython-json')
>>> upip.install('micropython-thread')
>>> upip.install('micropython-os')
```