weather_master

Master device description.

conf.json

Configuration file for run time variables. To apply any changes, restart the device.

```
{
    "ssid":"Rosko",
    "pass":"rando_password",
    "dssid":"E32Wtr-St",
    "dpass":"12345678",
    "gmt":"2",
    "useds":"1",
    "syncntp":"1",
    "log_delay":"600000",
    "lat":"42.6977",
    "lon":"23.3219"
}
```

- "ssid" - Nmae of SSID to connect to.
- "pass" - Password for that network.
- "dssid" - If a connection to "ssid" cant be established, create this network. so that slave device can connect to eat.
- "dpass" - Password for the default network.
- "gmt" - Greenwich Meridian Time offset in integer.
- "useds" - Adjust with daylight savings.
- "syncntp" - Attempt to sync with ntp.
- "log_delay" - Period of writing to log file.
- "lat" - Latitude for calculation of sunrise and sunset time.
- "lon" - Longitude for calculation of sunrise and sunset time.

mod.h

A template for creating new mod files.

storage_mod.h

Module with primitives for working with storage. Taken form here ESP32: Guide for MicroSD Card Module using Arduino IDE Each module has methods that follow this nomenclature:

{modname}_init is called to init the module and set it up {modname}_update is called on every update which is every 5 seconds

This carries between modules

API reference:

- listDir - List director's contents.
- createDir - Create a directory.
- removeDir - Remove directory.
- readFile - Display file contents on serial output.
- readFileToBuf - Read file contents into a buffer.
- writeFile - Write buffer to file.
- appendFile - Write at the end of file.
- renameFile - Rename file.
- deleteFile - Delete a file.
- testFileIO - File write throughput statistics.
- SD_STORAGE_init - Initialize sd card connection/

websrv_mod.h

Module for basic web server on the esp device. Consists of 2 pages, one is for configuration of the device, the other one is a real time dashboard that updates every 5 seconds.

- Configuration page
  - Most of the values of the json are reflected in conf.json
- Dashboard
  - Every 5 seconds an ajax request for each field on the dashboard.
  - For each parameter their is a endpoint.

astro_mod.h

Module for calculating basic astronomical data

- Need latitude and longitude for location
- Calculates the sunrise/sunset time using SunRise.h
- Calculates the moonrise/moonset time using MoonRise.h
- Calculates the moon phase(illumination and angle) using moonPhase.h

clock_mod.h

A module designed to interface with an ds3231 mini module (rtc) Also works with ntp. Optionally syncs with it once or can sync periodically, also an option. Has an automatic configurable adjustment for timezone and daylight savings

- Initializes the clock and syncs with ntp.
- On each update caches the current time. This is every 5 seconds. This is done just so we can have consistent time instead of reading the rtc every time. This is where the time adjustment happens, since the rtc module has no idea of a timezone and daylight savings.

config_mod.h

Module used for loading and storing the configuration. Using ArduinoJson.h library to read and write json. Everything is stored in static buffers. On every write to config it's written to file on sd card

display_mod.h

A modile for displaying some information to a display. Currently using a i2c 128x64 display but that can easily be changed. Every other module 'exports' some global variables, and this mdule displays them. Currently as a slideshow. The display runs asynchronously from the updating thread. Each 'slide' has a 5 second delay and clears the screen. The data from the modules is just formatted and then printed to display. Functions can easily be changed for a different display.

gas_mod.h

Measures gas particles with a MQ-7 sensor for carbon monoxide. Using MQ7Sensor.h to interface with it. On bootup it calibrates the sensor. Sensor can be re-calibrated durring runtime. That can be triggered from the web interface

logger_mod.h

Logging module. All the relevant informatuion is written to a static buffer, and when logging is done that is flushed to a logfile. The logging interval is not necessarily the same as the update interval. Update is done in intervals of 5 seconds, so as to have the freshest information, logging is done in the next 5 second interval. And the error is carried, so as to not shorten the interval overall. An offset of a few seconds makes no difference on a larger scale.

network_mod.h

Module that looks after the wireless network. If a connection cannot be established with the set network in the config json, it creates it's own. The network established also has credentials in the same json. It also hosts the website. The slave device looks for either the the preset network or the backup one. The espidf communication is mutually exclusive with wifi, except in the case they share a band. The network ssid's must match between those. Currently on the slave device are

hard-coded.

sl_device_mod.h

A module to read info from slave device. Currently it only works with one slave, But that can be extended. On receiving update from device, it caches it in its global variables.

st.h

Utility header for shared datatypes. Every module must include it.

temp_mod.h

Module to record readings from the BME280 sensor. Using the Adafruit_BME280 library. Reads and stores information.

zambretti_mod.h

This is a modified implementation of zambretti algorithm Modified to return bit field for the forecast.

```
enum WeatherState {
    Sunny     = 1 << 0,
    Rainy     = 1 << 1,
    Cloudy    = 1 << 2,
    Worsening = 1 << 3,
    Steady    = 1 << 4,
    Falling   = 1 << 5,
    Rising    = 1 << 6,
};
```

It combines both the pressure trend and the weather forecast

Since the time to produce a viable forecast is about 3 hours worth of readings, i keep the data in a file every time it reads. And pull info from there on restart. This is meant to guard against reboots. Any larger time would invalidate the data.

zambretti_mod_mk2.h

Another implementation of the zambretti algorithm adapted from here Also produces a bit field but a lot larger and more complex in terms of expressiveness. Otherwise uses linear interpolation to produce result

```
enum Pressure {
    Steady             = 0,
    Falling            = 1 << 0,
```

```
    Rising             = 1 << 1,
};
enum WeatherNow {
    Changable      = Rising << 1,
    Intermittent   = Rising << 2,
    Sunny          = Rising << 3,
    Rain           = Rising << 4,
    Stormy         = Rising << 5,
    Showery        = Rising << 6,
    Fine           = Rising << 7,
    Cloudy         = Rising << 8,
};
enum WeatherLater {
    Maybe              = Cloudy << 1,
    Intermittent_Later = Cloudy << 2,
    Sunny_Later        = Cloudy << 3,
    Rain_Later         = Cloudy << 4,
    Stormy_Later       = Cloudy << 5,
    Showery_Later      = Cloudy << 6,
    Fine_Later         = Cloudy << 7,
    Cloudy_Later       = Cloudy << 8,
};
enum UnSettled {
    Becoming           = Cloudy_Later << 5,
    Improving          = Cloudy_Later << 6,
    Settled            = Cloudy_Later << 7,
    Slightly           = Cloudy_Later << 8,
    Unsettled          = Cloudy_Later << 9,
    More               = Cloudy_Later << 10,
    Very               = Cloudy_Later << 11,
    Worse              = Cloudy_Later << 12,
    CANNOT_MAKE        = Cloudy_Later << 13,
};
```

The Pressure enum is for the pressure trend. The WeatherNow describes the weather now. The WeatherLater describes the weather later. The UnSettled describes how bad the weather goes. The names are in a loose order that would produce a coherent sentence most of the time. Two members of an enum are not always exclusive. Except the pressure, there can be only one from there. There is an array of strings to represent each field of the enum, so rendering is compact in terms of code and logic.

Since the time to produce a viable forecast is about 3 hours worth of readings, i keep the data in a file every time it reads. And pull info from there on restart. This is meant to guard against reboots. Any larger time would invalidate the data.

zambretti_meta_mod.h

A module to pick which implementation to use for the weather forecast algo.

weather_master.ino

Main file of the master device. Since all functionality is outsourced to modules the main thning this does is to init those modules, update info every 5 seconds, and dump current data to log file once in a while (defined interval).

weather_outside

weather_outside.ino

Similar to the temp_mod it also collects data from BME280 sensor. Also includes a analog uv light sensor. The uv 'strength' is the voltage if the output pin and is also linear. Using a GUVA-S12SD sensor. Sends those readings to the master device in the following structure:

```
typedef struct struct_message {
   float temp;
   float hum;
   float pres;
   float uv_idx;
} struct_message;
```

There has to be parity on the master device if we want to get the correct info.

The module looks for the wifi network if the master device. If it cant find it it tries to see if the backup network is up. It will attempt to reset the network connection if device is not found