

JSON parser and emitter for C/C++

Features

- ISO C and ISO C++ compliant portable code
- Very small footprint
- No dependencies
- `json_scanf()` scans a string directly into C/C++ variables
- `json_printf()` prints C/C++ variables directly into an output stream
- `json_setf()` modifies an existing JSON string
- `json_fread()` reads JSON from a file
- `json_fprintf()` writes JSON to a file
- Built-in base64 encoder and decoder for binary data
- Parser provides low-level callback API and high-level scanf-like API
- 100% test coverage
- Used in [Mongoose OS](#), an operating system
for connected commercial products on low-power microcontrollers

API reference

`json_scanf()`, `json_vscanf`

```
int json_scanf(const char *str, int str_len, const char *fmt, ...);
int json_vscanf(const char *str, int str_len, const char *fmt, va_list ap);

/* json_scanf's %M handler */
typedef void (*json_scanner_t)(const char *str, int len, void *user_data);
```

Scans the JSON string `str`, performing scanf-like conversions according to `fmt`.

`fmt` uses `scanf()`-like format, with the following differences:

1. Object keys in the format string don't have to be quoted, e.g. "{key: %d}"
2. Order of keys in the format string does not matter, and the format string may omit keys to fetch only those that are of interest, for example,
assume `str` is a JSON string `{ "a": 123, "b": "hi", c: true }`.
We can fetch only the value of the `c` key:

```
int value = 0;
json_scanf(str, strlen(str), "{c: %B}", &value);
```

3. Several extra format specifiers are supported:
 - `%B`: consumes `int *` (or `char *`, if `sizeof(bool) == sizeof(char)`), expects boolean `true` or `false`.
 - `%Q`: consumes `char **`, expects quoted, JSON-encoded string. Scanned string is malloc-ed, caller must free() the string.
 - `%V`: consumes `char **`, `int *`. Expects base64-encoded string.
Result string is base64-decoded, malloced and NUL-terminated.
The length of result string is stored in `int *` placeholder.
Caller must free() the result.
 - `%H`: consumes `int *, char **`.

Expects a hex-encoded string, e.g. "fa014f".

Result string is hex-decoded, malloced and NUL-terminated.

The length of the result string is stored in `int *placeholder`.

Caller must free() the result.

- %M: consumes custom scanning function pointer and

`void *user_data` parameter - see `json_scanner_t` definition.

- %T: consumes `struct json_token *`, fills it out with matched token.

Returns the number of elements successfully scanned & converted.

Negative number means scan error.

Example - scan arbitrary JSON string:

```
// str has the following JSON string (notice keys are out of order):
// { "a": 123, "d": true, "b": [1, 2], "c": "hi" }

int a = 0, d = 0;
char *c = NULL;
void *my_data = NULL;
json_scanf(str, strlen(str), "{ a:%d, b:%M, c:%Q, d:%B }",
           &a, scan_array, my_data, &c, &d);

// This function is called by json_scanf() call above.
// str is "[1, 2]", user_data is my_data.
static void scan_array(const char *str, int len, void *user_data) {
    struct json_token t;
    int i;
    printf("Parsing array: %.s\n", len, str);
    for (i = 0; json_scanf_array_elem(str, len, "", i, &t) > 0; i++) {
        printf("Index %d, token [%.*s]\n", i, t.len, t.ptr);
    }
}
```

Example - parse array of objects:

```
// str has the following JSON string - array of objects:
// { "a": [ {"b": 123}, {"b": 345} ] }
// This example shows how to iterate over array, and parse each object.

int i, value, len = strlen(str);
struct json_token t;

for (i = 0; json_scanf_array_elem(str, len, ".a", i, &t) > 0; i++) {
    // t.type == JSON_TYPE_OBJECT
    json_scanf(t.ptr, t.len, "{b: %d}", &value); // value is 123, then 345
}
```

json_scanf_array_elem()

```
int json_scanf_array_elem(const char *s, int len,
                           const char *path,
                           int index,
                           struct json_token *token);
```

A helper function to scan an array item with given path and index.

Fills `token` with the matched JSON token.

Returns 0 if no array element found, otherwise non-0.

json_printf()

Frozen printing API is pluggable. Out of the box, Frozen provides a way

to print to a string buffer or to an opened file stream. It is easy to tell Frozen to print to another destination, for example, to a socket, etc. Frozen does this by defining an "output context" descriptor which has a pointer to a low-level printing function. If you want to print to another destination, just define your specific printing function and initialise output context with it.

This is the definition of the output context descriptor:

```
struct json_out {
    int (*printer)(struct json_out *, const char *str, size_t len);
    union {
        struct {
            char *buf;
            size_t size;
            size_t len;
        } buf;
        void *data;
        FILE *fp;
    } u;
};
```

Frozen provides two helper macros to initialise two built-in output descriptors:

```
struct json_out out1 = JSON_OUT_BUF(buf, len);
struct json_out out2 = JSON_OUT_FILE(fp);
```

```
typedef int (*json_printf_callback_t)(struct json_out *, va_list *ap);
int json_printf(struct json_out *, const char *fmt, ...);
int json_vprintf(struct json_out *, const char *fmt, va_list ap);
```

Generate formatted output into a given string buffer, auto-escaping keys.

This is a superset of printf() function, with extra format specifiers:

- %B print json boolean, true or false. Accepts an int.
- %Q print quoted escaped string or null. Accepts a const char *.
- %.*Q same as %Q, but with length. Accepts int, const char *
- %V print quoted base64-encoded string. Accepts a const char *, int.
- %H print quoted hex-encoded string. Accepts a int, const char *.
- %M invokes a json_printf_callback_t function. That callback function can consume more parameters.

Return number of bytes printed. If the return value is bigger than the supplied buffer, that is an indicator of overflow. In the overflow case, overflow bytes are not printed.

Example:

```
json_printf(&out, "{%Q: %d, x: [%B, %B], y: %Q}", "foo", 123, 0, -1, "hi");
// Result:
// {"foo": 123, "x": [false, true], "y": "hi"}
```

To print a complex object (for example, serialise a structure into an object), use %M format specifier:

```
struct my_struct { int a, b; } mys = {1,2};
json_printf(&out, "{foo: %M, bar: %d}", print_my_struct, &mys, 3);
// Result:
// {"foo": {"a": 1, "b": 2}, "bar": 3}
```

```

int print_my_struct(struct json_out *out, va_list *ap) {
    struct my_struct *p = va_arg(*ap, struct my_struct *);
    return json_printf(out, "{a: %d, b: %d}", p->a, p->b);
}

```

json_printf_array()

```
int json_printf_array(struct json_out *, va_list *ap);
```

A helper %M callback that prints contiguous C arrays.

Consumes void *array_ptr, size_t array_size, size_t elem_size, char *fmt

Returns number of bytes printed.

json_walk() - low level parsing API

```

/* JSON token type */
enum json_token_type {
    JSON_TYPE_INVALID = 0, /* memsetting to 0 should create INVALID value */
    JSON_TYPE_STRING,
    JSON_TYPE_NUMBER,
    JSON_TYPE_TRUE,
    JSON_TYPE_FALSE,
    JSON_TYPE_NULL,
    JSON_TYPE_OBJECT_START,
    JSON_TYPE_OBJECT_END,
    JSON_TYPE_ARRAY_START,
    JSON_TYPE_ARRAY_END,
    JSON_TYPES_CNT,
};

/*
 * Structure containing token type and value. Used in `json_walk()` and
 * `json_scanf()` with the format specifier `%T` .
 */
struct json_token {
    const char *ptr;           /* Points to the beginning of the value */
    int len;                  /* Value length */
    enum json_token_type type; /* Type of the token, possible values are above */
};

/* Callback-based API */
typedef void (*json_walk_callback_t)(void *callback_data,
                                      const char *name, size_t name_len,
                                      const char *path,
                                      const struct json_token *token);

/*
 * Parse `json_string`, invoking `callback` in a way similar to SAX parsers;
 * see `json_walk_callback_t` .
 */
int json_walk(const char *json_string, int json_string_length,
              json_walk_callback_t callback, void *callback_data);

```

`json_walk()` is a low-level, callback based parsing API.

`json_walk()` calls a given callback function for each scanned value.

Callback receives a name, a path to the value, a JSON token that points to the value and an arbitrary user data pointer.

The path is constructed using this rule:

- Root element has "" (empty string) path

- When an object starts, . (dot) is appended to the path
- When an object key is parsed, a key name is appended to the path
- When an array is parsed, an [ELEMENT_INDEX] is appended for each element

For example, consider the following json string:

```
{ "foo": 123, "bar": [ 1, 2, { "baz": true } ] }.
```

The sequence of callback invocations will be as follows:

- type: JSON_TYPE_OBJECT_START, name: NULL, path: "", value: NULL
- type: JSON_TYPE_NUMBER, name: "foo", path: ".foo", value: "123"
- type: JSON_TYPE_ARRAY_START, name: "bar", path: ".bar", value: NULL
- type: JSON_TYPE_NUMBER, name: "0", path: ".bar[0]", value: "1"
- type: JSON_TYPE_NUMBER, name: "1", path: ".bar[1]", value: "2"
- type: JSON_TYPE_OBJECT_START, name: "2", path: ".bar[2]", value: NULL
- type: JSON_TYPE_TRUE, name: "baz", path: ".bar[2].baz", value: "true"
- type: JSON_TYPE_OBJECT_END, name: NULL, path: ".bar[2]", value: "{ \"baz\": true }"
- type: JSON_TYPE_ARRAY_END, name: NULL, path: ".bar", value: "[1, 2, { \"baz\": true }]"
- type: JSON_TYPE_OBJECT_END, name: NULL, path: "", value: "{ \"foo\": 123, \"bar\": [1, 2, { \"baz\": true }] }"

If top-level element is an array: [1, {"foo": 2}]

- type: JSON_TYPE_ARRAY_START, name: NULL, path: "", value: NULL
- type: JSON_TYPE_NUMBER, name: "0", path: "[0]", value: "1"
- type: JSON_TYPE_OBJECT_START, name: "1", path: "[1]", value: NULL
- type: JSON_TYPE_NUMBER, name: "foo", path: "[1].foo", value: "2"
- type: JSON_TYPE_OBJECT_END, name: NULL, path: "[1]", value: "{\"foo\": 2}"
- type: JSON_TYPE_ARRAY_END, name: NULL, path: "", value: "[1, {\"foo\": 2}]"

If top-level element is a scalar: true

- type: JSON_TYPE_TRUE, name: NULL, path: "", value: "true"

json_fprintf(), json_vfprintf()

```
/*
 * Same as json_printf, but prints to a file.
 * File is created if does not exist. File is truncated if already exists.
 */
int json_fprintf(const char *file_name, const char *fmt, ...);
int json_vfprintf(const char *file_name, const char *fmt, va_list ap);
```

json_asprintf(), json_vasprintf()

```
/*
 * Print JSON into an allocated 0-terminated string.
 * Return allocated string, or NULL on error.
 * Example:
 *
 * ```` c
 *     char *str = json_asprintf("{a:%H}", 3, "abc");
 *     printf("%s\n", str); // Prints "616263"
 *     free(str);
 * ````
 */
char *json_asprintf(const char *fmt, ...);
char *json_vasprintf(const char *fmt, va_list ap);
```

json_fread()

```
/*
 * Read the whole file in memory.
 * Return malloc-ed file content, or NULL on error. The caller must free().
 */
char *json_fread(const char *file_name);
```

json_setf(), json_vsetf()

```
/*
 * Update given JSON string `s,len` by changing the value at given `json_path`.
 * The result is saved to `out` . If `json_fmt` == NULL, that deletes the key.
 * If path is not present, missing keys are added. Array path without an
 * index pushes a value to the end of an array.
 * Return 1 if the string was changed, 0 otherwise.
 *
 * Example: s is a JSON string { "a": 1, "b": [ 2 ] }
 *   json_setf(s, len, out, ".a", "7");      // { "a": 7, "b": [ 2 ] }
 *   json_setf(s, len, out, ".b", "7");      // { "a": 1, "b": 7 }
 *   json_setf(s, len, out, ".b[]", "7");    // { "a": 1, "b": [ 2,7 ] }
 *   json_setf(s, len, out, ".b", NULL);     // { "a": 1 }
 */
int json_setf(const char *s, int len, struct json_out *out,
              const char *json_path, const char *json_fmt, ...);

int json_vsetf(const char *s, int len, struct json_out *out,
               const char *json_path, const char *json_fmt, va_list ap);
```

json_pretty()

```
/*
 * Pretty-print JSON string `s,len` into `out` .
 * Return number of processed bytes in `s` .
 */
int json_pretty(const char *s, int len, struct json_out *out);
```

json_pretty_file()

```
/*
 * Prettify JSON file `file_name` .
 * Return number of processed bytes, or negative number of error.
 * On error, file content is not modified.
 */
int json_pretty_file(const char *file_name);
```

json_next_key(), json_next_elem()

```

/*
 * Iterate over an object at given JSON `path`.
 * On each iteration, fill the `key` and `val` tokens. It is OK to pass NULL
 * for `key` , or `val` , in which case they won't be populated.
 * Return an opaque value suitable for the next iteration, or NULL when done.
 *
 * Example:
 *
 * ````c
 * void *h = NULL;
 * struct json_token key, val;
 * while ((h = json_next_key(s, len, h, ".foo", &key, &val)) != NULL) {
 *     printf("[%.*s] -> [%.*s]\n", key.len, key.ptr, val.len, val.ptr);
 * }
 * ```
 */
void *json_next_key(const char *s, int len, void *handle, const char *path,
                    struct json_token *key, struct json_token *val);

/*
 * Iterate over an array at given JSON `path` .
 * Similar to `json_next_key` , but fills array index `idx` instead of `key` .
 */
void *json_next_elem(const char *s, int len, void *handle, const char *path,
                     int *idx, struct json_token *val);

```

Minimal mode

By building with `-DJSON_MINIMAL=1` footprint can be significantly reduced.

The following limitations apply in this configuration:

- Only integer numbers are supported. This affects parsing and `%f/%lf` conversions in `printf` and `scanf`.
- Hex ('%H') and base64 (%V) conversions are disabled.

Examples

Print JSON configuration to a file

```

json_fprintf("settings.json", "{ a: %d, b: %Q }", 123, "string_value");
json_prettyfile("settings.json"); // Optional

```

Read JSON configuration from a file

```

struct my_config { int a; char *b; } c = { .a = 0, .b = NULL };
char *content = json_fread("settings.json");
json_scanf(content, strlen(content), "{a: %d, b: %Q}", &c.a, &c.b);

```

Modify configuration setting in a JSON file

```
const char *settings_file_name = "settings.json", *tmp_file_name = "tmp.json";
char *content = json_fread(settings_file_name);
FILE *fp = fopen(tmp_file_name, "w");
struct json_out out = JSON_OUT_FILE(fp);
json_setf(content, strlen(content), &out, ".b", "%Q", "new_string_value");
fclose(fp);
json_prettyify_file(tmp_file_name); // Optional
rename(tmp_file_name, settings_file_name);
```

Contributions

To submit contributions, sign

[Cesanta CLA](#)

and send GitHub pull request.

Licensing

Frozen is released under the

[Apache 2.0 license](#).

For commercial support and professional services,
[contact us](#).

Mongoose OS and Microsoft Azure IoT

This tutorial demonstrates the following:

- How to build a skeleton for a commercial smart device, managed using Microsoft Azure IoT
- How to reuse the code for different hardware platforms
- How to configure devices via Azure IoT Hub device twin
- How to perform bulk over-the-air (OTA) updates using [Azure IoT Hub automatic device management](#)

Setup Azure IoT Hub

- Create an account on [portal.azure.com](#)
- Install `az`, the Azure command line utility, by [following these instructions](#)
- Start command prompt (or terminal on UNIX/Mac), login and follow the instructions:

```
az login
```

- Create an IoT hub. Note:
 - Use EU-West region (this is important, since not all regions have IoT device configuration enabled)
 - Choose standard pricing tier

Setup device

- Pick one of the supported devices. We suggest to choose from [recommended devboards](#)
- Connect your device to your workstation via USB
- Complete [Quickstart Guide](#) steps 1-7 inclusive.
As a result, your device should be connected to the Internet
- Provision your device to Azure IoT with a single command:

```
mos azure-iot-setup --azure-hub-name YOUR_AZURE_HUB_NAME
```

This command performs certificate management for you, and onboards your device on a cloud. If your device has an [Atmel ECC508A](#) secure element attached, Mongoose OS automatically uses ECC508A chip for TLS and keeps your credentials securely stored.

A newly provisioned device will appear in the IoT hub's device list. On the picture, an ESP8266 board was used. You will get a different device ID, depending on the hardware platform you're using.

Configuring the device using Azure IoT Hub device twin

In the IoT Hub blade of the Azure portal, click on the device ID shown in the device list, then select the "Device twin" tab:

That will bring up the device twin editor:

The firmware that we've build following quickstart guide, contains

code that synchronises device LED with the `desired.on` twin setting.

In the device twin editor, add `desired.on` boolean key:

```
...
"desired": {
  "on": true,    // <-- add this
  ...
},
...
```

Set it to `true` or

`false`, and observe how the device reacts by switching the LED on/off:



This example demonstrates remote device configuration using the device twin, and can be applied to a broad range of use cases.

IoT button

Open `fs/init.js` file in your favorite editor, copy/paste the following code:

```

load('api_config.js');
load('api_gpio.js');
load('api_sys.js');
load('api_mqtt.js');

let pin = Cfg.get('board.btn1.pin'); // User button GPIO pin
let topic = 'devices/' + Cfg.get('device.id') + '/messages/events/';

let f = function() {
  let message = JSON.stringify({
    total_ram: Sys.total_ram(),
    free_ram: Sys.free_ram(),
    uptime: Sys.uptime(),
  });
  let ok = MQTT.pub(topic, message, 1);
  print('Published:', ok, topic, '->', message);
};

GPIO.set_button_handler(pin, GPIO.PULL_UP, GPIO.INT_EDGE_NEG, 20, f, null);

```

When done, copy `fs/init.js` to the device and reboot the device:

```

mos put fs/init.js
mos call Sys.Reboot

```

Then, press a button to send an MQTT message.

Bulk OTA updates with the Azure IoT Hub automatic device management feature

Register a couple more devices in the Azure IoT Hub, following the same steps as in the Setup Device section above:

```
mos azure-iot-setup --azure-hub-name YOUR_AZURE_HUB_NAME
```

For each device, edit the Device Twin adding the following snippet:

```

"tags": {
  "city": "dublin"
},

```

Make some change in the firmware code. Replace the code in `fs/init.js` with
`print('hello new version');` and rebuild the firmware:

```

mos build --platform YOUR_PLATFORM
mos console

```

This will generate a new firmware `build/fw.zip`. Upload this file to some web server of your choice.

In the Azure IoT Hub blade go to the Automatic Device management section and create a new device configuration:

Give this configuration a name. On step 2, specify twin settings you'd like to apply: "Device Twin Path" set to `properties.desired`, and "Content" to

```

{
  "ota": {
    "url": "URL_OF_THE_REBUILT_fw.zip"
  }
}

```

Skip Step 3, and on Step 4 specify "Target Condition":

`tags.city='dublin'`. Submit configuration.

Open the device twin of any device, make sure that the `desired.properties.ota.url` attribute has been applied. The firmware we've built uses the [ota-shadow](#) library which observes that attribute in the twin, and starts an OTA update if it is changes.

The OTA parameters are reported by the firmware in the `properties.reported.ota`, where you can see various metrics:

```
"reported": {  
  "ota": {  
    "fw_version": "1.0",  
    "fw_id": "20180716-150324/lsm@23146fe+",  
    "mac": "5ECF7F060046",  
    "device_id": "esp8266_060046",  
    "app": "demo-js",  
    "message": "progress",  
    "status": 1,  
    "is_committed": true,  
    "commit_timeout": 600,  
    "partition": 0,  
    "progress_percent": 27,  
    ...  
  },
```

Note the `progress_percent` value. If you refresh the twin editor, it'll change to indicate the OTA update progress. When the update is complete, the value of `ota.fw_id` changes to indicate the new timestamp of the firmware.

Observe console statement, note the `Hello new version` message.

You have just performed an over-the-air firmware update accross multiple devices seamlessly leveraging Azure IoT.

Mongoose OS + private MQTT server

The `mqtt` library provides [MQTT protocol](#) API that allows devices to talk to MQTT servers.

Mongoose OS implements MQTT 3.1.1 client functionality, and works with all popular MQTT server implementations, like AWS IoT, Google IoT Core, Microsoft Azure, IBM Watson, HiveMQ, Mosquitto, etc.

Usage

In order to use MQTT functionality,

- Include `mqtt` library in the `libs` section of the `mos.yml` file of your app:

```
libs:  
  ...  
  - origin: https://github.com/mongoose-os-libs/mqtt # <-- Add this!
```

- Rebuild and reflash your app:

```
mos build --platform YOUR_PLATFORM # e.g. stm32, esp32, cc3220, esp8266, cc3200, etc  
mos flash
```

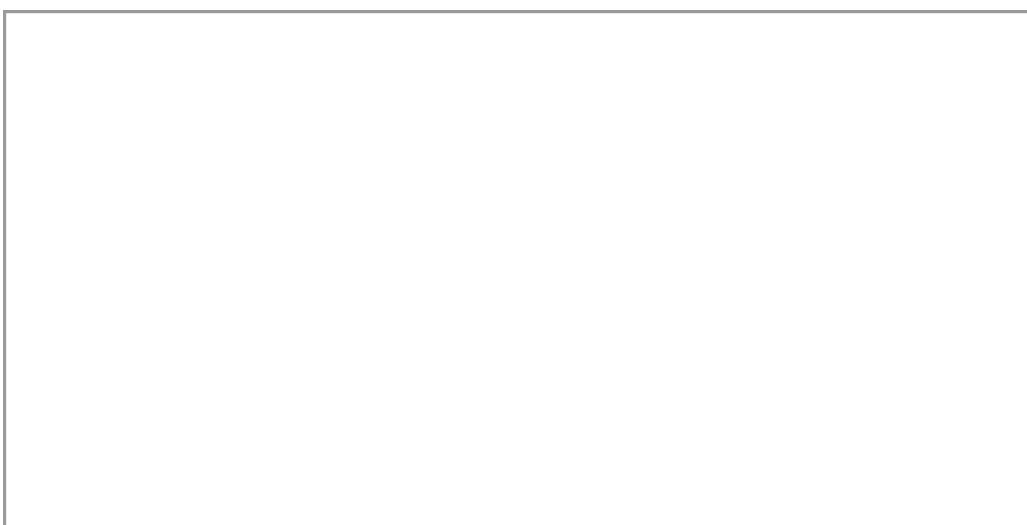
NOTE: if you're using a pre-built Mongoose OS demo-js app, that step is not required, since mqtt library is already included in the demo-js.

In order to talk to an MQTT server, configure MQTT server settings - see Configuration section below. Once configured, Mongoose OS keeps that connection alive by reconnecting and re-subscribing to all topics after disconnections - you do not need to implement the reconnection logic.

```
$ mos config-set mqtt.enable=true mqtt.server=my_server:1883
```

If you want to use TLS, set mqtt.ssl_ca_cert=ca.pem. Make sure that ca.pem file has required CA certificates. If you want to use mutual TLS, set mqtt.ssl_cert=CLIENT_CERT.pem and mqtt.ssl_key=PRIVATE_KEY.pem.

See example video (don't forget to set mqtt.enable=true before you try it):



MQTT publish and subscribe - JavaScript

```
load('api_mqtt.js');  
load('api_gpio.js');  
  
let pin = 0, topic = 'my/topic';  
  
GPIO.set_button_handler(pin, GPIO.PULL_UP, GPIO.INT_EDGE_NEG, 200, function() {  
  MQTT.pub('my/topic', JSON.stringify({ a: 1, b: 2 }));  
}, null);  
  
MQTT.sub('my/topic/#', function(conn, topic, msg) {  
  print('Topic:', topic, 'message:', msg);  
}, null);
```

MQTT publish and subscribe - C/C++

```

#include "mgos_mqtt.h"

static void handler(struct mg_connection *c, const char *topic, int topic_len,
                    const char *msg, int msg_len, void *userdata) {
    LOG(LL_INFO, ("Got message on topic %.%s", topic_len, topic));
    ...
}

mgos_mqtt_sub("my/#", handler, NULL);      /* Subscribe */
mgos_mqtt_pub("my/topic", "hi", 2, 1, 0);   /* Publish */

```

Configuration Reference

The MQTT library adds `mqtt` section to the device configuration:

```
{
  "clean_session": true,           // Clean session info stored on server
  "client_id": "",                // If not set, device.id is used
  "enable": false,                 // Enable MQTT functionality
  "keep_alive": 60,                // How often to send PING messages in seconds
  "pass": "",                     // User password
  "reconnect_timeout_max": 60,     // Maximum reconnection timeout in seconds
  "reconnect_timeout_min": 2,       // Minimum reconnection timeout in seconds
  "server": "iot.eclipse.org:1883", // SERVER:PORT to connect to
  "ssl_ca_cert": "",              // Set this to file name with CA certs to enable TLS
  "ssl_cert": "",                 // Client certificate for mutual TLS
  "ssl_cipher_suites": "",        // TLS cipher suites
  "ssl_key": "",                  // Private key for the client certificate
  "ssl_psk_identity": "",         // If set, a preshared key auth is used
  "ssl_psk_key": "",              // Preshared key
  "user": "",                     // MQTT user name, if MQTT auth is used
  "will_message": "",             // MQTT last will message
  "will_topic": ""                // MQTT last will topic
}
```

Mongoose OS + Google IoT Core

This tutorial demonstrates the following:

- How to build a skeleton firmware for a commercial smart device, managed by Google IoT Core
- How to reuse the code for different hardware platforms
- How to control devices via Google IoT Core config object
- How to report device state via Google IoT Core state object

See following video that repeats setup steps mentioned below:



Setup Google IoT Core

- Install [gcloud command line tool](#)
- Authenticate with Google Cloud:

```
gcloud auth application-default login
```

- Create cloud project - choose your unique project name:

```
gcloud projects create YOUR_PROJECT_NAME
```

- Add permissions for IoT Core:

```
gcloud projects add-iam-policy-binding YOUR_PROJECT_NAME --member=serviceAccount:cloud-iot@system.gserviceaccount.com --role=roles/pubsub.publish
```

- Set default values for gcloud:

```
gcloud config set project YOUR_PROJECT_NAME
```

- Create PubSub topic for device data:

```
gcloud pubsub topics create iot-topic
```

- Create PubSub subscription for device data:

```
gcloud pubsub subscriptions create --topic iot-topic iot-subscription
```

- Create device registry:

```
gcloud iot registries create iot-registry --region europe-west1 --event-notification-config=topic=iot-topic
```

Setup device

- Pick one of the supported devices. We suggest to choose from [recommended devboards](#)
- Connect your device to your workstation via USB
- Complete [Quickstart Guide](#) steps 1-7 inclusive.
As a result, your device should be connected to the Internet
- Get project ID of your new project:

```
gcloud projects list
```

- Register device on Google IoT Core. If a device is already registered, this command deletes it, then registers again. Note that this command is using `YOUR_PROJECT_ID` instead of `YOUR_PROJECT_NAME`. Take the project ID from the result of your previous command:

```
mos gcp-iot-setup --gcp-project YOUR_PROJECT_ID --gcp-region europe-west1 --gcp-registry iot-registry
```

This command performs certificate management for you, and onboards your device on a cloud. If your device has an

[Atmel ECC508A](#) secure element

attached, Mongoose OS automatically uses ECC508A chip for TLS and keeps your credentials securely stored.

Controlling LED via Google IoT Core

Open `fs/init.js` in your favorite editor, copy-paste the following snippet and save:

```
load('api_config.js');
load('api_gpio.js');
load('api_mqtt.js');

let led = Cfg.get('board.led1.pin');
let topic = '/devices/' + Cfg.get('device.id') + '/config';

GPIO.set_mode(led, GPIO.MODE_OUTPUT);

MQTT.sub(topic, function(conn, topic, msg) {
  print('Topic:', topic, 'message:', msg);
  let obj = JSON.parse(msg) || {led: 0};
  GPIO.write(led, obj.led);
}, null);
```

The snippet above subscribes to the `/config` notifications.

Once the config object is sent to device by Google IoT Core,
the handler function is called and lights the corresponding LED.

In the command prompt (or terminal on Linux/Mac), enter the following commands
to copy `init.js` to the device, reboot the device, and start monitoring
serial output:

```
mos put fs/init.js
mos call Sys.Reboot
mos console
```



Find a registered device in a Google IoT Core device registry.

Click on "Update Config" button, and in the opened dialog, enter
the following configuration and hit "SEND TO DEVICE":

```
{"led": 1}
```

Note how one of the LEDs turns on. NOTE: some dev boards, notably Espressif,
have LED pins inverted, i.e. they switch off on value `1` and switch on,
on value `0`. Enter

```
{"led": 0}
```

Note how the LED turns off.

Important thing to note: Google IoT Core send device config object
each time device comes online or a config change is made. That means,
that if you power cycle the device, it'll sync its LED state with the
cloud as soon as it gets connected.

Reporting state to Google IoT Core

We are going to report free RAM to the Google IoT Core periodically, to the Google's special /state object.

Open fs/init.js in your favorite editor, copy-paste the following snippet and save:

```
load('api_config.js');
load('api_mqtt.js');
load('api_sys.js');
load('api_timer.js');

let topic = '/devices/' + Cfg.get('device.id') + '/state';

Timer.set(3000 /* milliseconds */, Timer.REPEAT, function() {
  let msg = JSON.stringify({free: Sys.free_ram(), total: Sys.total_ram()});
  print(topic, '->', msg);
  MQTT.pub(topic, msg, 1);
}, null);
```



In the command prompt (or terminal on Linux/Mac), enter the following commands to copy init.js to the device, reboot the device, and start monitoring serial output:

```
mos put fs/init.js
mos call Sys.Reboot
mos console
```

In the Google IoT Web console, click on the "Configuration and state history", and see how new state objects are arriving. Click on any state object, select "text" view, and check the JSON string that is sent by device.

Using ATECC508A crypto chip

See <https://mongoose-os.com/blog/mongoose-os-google-iot-ecc508a/> on how to use ATECC508A crypto chip with Mongoose OS and Google IoT Core.

Sending debug logs to stackdriver

It may be very useful for remote debugging to be able to see a device's logs in Stackdriver. To do so you need to configure the debug.stdout_topic / debug.stderr_topic fields, add a topic in Google IoT, create a cloud function that listens to the topic and writes a new log entry that you can watch live in Stackdriver. We'll be using the topic name of 'debug', and associating both debug and error logs with it, but you can use whatever you like!

*** Warning *** This may incur a *lot* of storage and processing costs running full debug through the cloud functions and storage. Proceed at own risk.

Configure device

It's recommended to set it via a config{7/8/9}.json file that you load onto the device, or you can simply configure the device via mos tool:

```
mos call Config.Set '{"config": {"debug": {"stdout_topic": "/devices/{YOURDEVICEID}/events/debug", "stderr_topic": "/devices/{YOURDEVICEID}/events/debug"} }}'
```

Replace {YOURDEVICEID} with the ID of the device.

Add a topic

- Log into your cloud console, navigate to IoT core and choose your registry.
- Click add/edit topics
- There's an add topic and subfolder button, click that
- Choose 'add topic' from the popup
- Fill the topic and subfolder fields with 'debug'
- Go back to the add/edit topics screen, scroll down and click update

Create a Firebase cloud function

We presume you know how to use firebase cloud functions, and we'll be assuming you have a local functions folder that you're editing and uploading. Here's some docu if you need to get up to speed: <https://firebase.google.com/docs/functions/get-started>

This is our function to intercept the debug topic and write a meaningful log result to Stackdriver.

```
var functions = require('firebase-functions');
const {Logging} = require('@google-cloud/logging');

// create the Stackdriver Logging client
const logging = new Logging({
    projectId: process.env.GCLOUD_PROJECT,
});

// start cloud function

exports.deviceLog =
    functions.pubsub.topic('debug').onPublish((message) => {

        //console.log("message received: ", message);

        const log = logging.log('debug-logs');
        const metadata = {
            // Set the Cloud IoT Device you are writing a log for
            // you extract the required device info from the PubSub attributes
            resource: {
                type: 'cloudiot_device',
                labels: {
                    project_id: message.attributes.projectId,
                    device_num_id: message.attributes.deviceNumId,
                    device_registry_id: message.attributes.deviceRegistryId,
                    location: message.attributes.location,
                }
            },
            labels: {
                // note device_id is not part of the monitored resource, but you can
                // include it as another log label
                device_id: message.attributes.deviceId,
            }
        };

        let buff = new Buffer(message.data, 'base64');
        let text = buff.toString('ascii');
        //console.log("data: ", text);

        let textArray = text.split("|");
        if (textArray[0].substr(-1) === "2"){
            metadata['severity'] = 'ERROR';
        } else {
            metadata['severity'] = 'DEBUG';
        }

        // write the log entryto Stackdriver Logging
        const entry = log.entry(metadata, textArray[1]);
        return log.write(entry);
});
```

Deploy that function and then go to the Stackdriver console, and drill down to the function to check that it's executing without errors.

View the logs

Within Stackdriver click the resource filter and choose Cloud IoT Device, and you can either choose all device num or a specific (Google ID) device to see the nicely formatted debug logs.

Samsung Artik Cloud

In order to use Samsung Artik Cloud service, a "Generic MQTT server" library is required. The sequence of steps to onboard with Artik Cloud is outlined below.

Prerequisites:

- ESP8266 NodeMCU or ESP32 development boards
- Samsung Artik Cloud account

Onboarding sequence

Install Mongoose OS

Run `mos` tool, install Mongoose OS and set up WiFi

Login to Artik Cloud and create a device with type DHT11

Generate device token

Click on the device, generate device token

Configure device

Open device configuration in the Web UI, click on Expert View, and set the following fields (use your specific device ID and token):

- `mqtt.enable=true`
- `mqtt.server=api.artik.cloud:8883`
- `mqtt.user=YOUR_DEVICE_ID`
- `mqtt.pass=YOUR_DEVICE_TOKEN`
- `mqtt.ssl_ca_cert=ca.pem`

Make device report data

Open Device Files tab, click on `init.js` file. Paste this code that sends fake temperature/humidity data to the cloud:

```

load('api_config.js');
load('api_mqtt.js');
load('api_sys.js');
load('api_timer.js');

let sdid = Cfg.get('mqtt.user');
let topic = '/v1.1/messages/' + sdid;

Timer.set(1000, true, function() {
  let data = {temperature:24, humidity:Sys.free_ram()};
  let ok = MQTT.pub(topic, JSON.stringify(data), 1);
  print('Published', ok, JSON.stringify(data));
}, null);

```

Click 'Save + Reboot' button

Graph reported data

Click on CHARTS tab in Artik Cloud console.

Add graph for DHT11 device, humidity - in reality, that reports free RAM.

Summary

The process above shows how to use Samsung Artik Cloud to authorise devices and report data via MQTT.

IBM Watson

This is a quick start tutorial for [IBM Watson IoT Platform](#) for Mongoose OS.

Quick Setup

- Complete [Quickstart Guide](#) steps 1-7 inclusive.
As a result, your device should be connected to the Internet
- Provision your device to IBM Watson:

```
mos watson-iot-setup
```

This does not require any credentials and will set up the device for [QuickStart cloud dashboard](#) connectivity.

With QuickStart dashboard, you will be able to send data from the device at a limited rate but device management (sending commands to the device) is not available.

Note: in order to make a button press to send events,

- Mongoose OS RPC support should be disabled by

```
mos config-set rpc.mqtt.enable=false
```
- A default topic name should be changed. Open `fs/init.js` and edit `set_button_handler()` callback by changing topic name:

```
let topic = 'iot-2/evt/mos/fmt/json';
```

Save, upload to a device using `mos put fs/init.js`, and reboot.

IBM Cloud Account setup

If you already have an IBM Cloud account and a cloud orgaization set up (you can register for a free trial [here](#)) you can use `mos` to add a

device to your org.

You will need to obtain app access credentials for `mos` to use:

- In the left side pane of your Watson IoT Platform dashboard select *Apps*
- Click the *Generate API Key* button in the upper right corner
- For description, enter any meaningful text - e.g. "MOS CLI tool" and click *Next*
- For *Role* select *Standard Application*
- Copy the *API Key* and *Authentication Token* and use them on the `mos` command line:

```
$ mos watson-iot-setup --watson-api-key a-bcdefgh-qwertio --watson-api-auth-token "tokengoeshere"
Using port /dev/ttyUSB0
Connecting to the device...
esp8266 1AFE34A5930F running demo-c
Org ID: bcdefgh
Device ID: esp8266_A5930F
Checking device type "mos"...
Creating device "esp8266_A5930F"...

Updating config:
device.id = esp8266_A5930F
watson.client_id = d:bcdefgh:mos:esp8266_A5930F
watson.enable = true
watson.host_name = bcdefgh.messaging.internetofthings.ibmcloud.com
watson.token = SECRET
Setting new configuration...
Saving and rebooting...
```

Note: As with other flags, values can be specified as environment variables: `MOS_WATSON_API_KEY` and `MOS_WATSON_API_AUTH_TOKEN`.

Invoked with no additional flags, as above, `mos` will use device type `mos` (will be created if necessary) and device ID derived from platform and the device's unique ID (usually MAC address) and a random auth token. Additional flags can be provided to override these defaults:

- `--watson-device-type` - specifies device type
- `--watson-device-id` - specifies device id
- `--watson-device-auth-token` - specifies device auth token instead of generating a random one
- `--watson-api-host-name` - use this host name for API calls instead of `ORG_ID.internetofthings.ibmcloud.com`
- `--watson-messaging-host-name` - use this host name for device connections instead of `ORG_ID.messaging.internetofthings.ibmcloud.com`

Mongoose OS + AWS IoT

This tutorial demonstrates the following:

- How to build a skeleton for a commercial smart device, managed by AWS IoT
- Internet Button in 2 minutes
- How to control devices via AWS IoT device shadow
- How to report metrics data via AWS IoT
- How to perform OTA updates using AWS IoT device shadow

Setup AWS IoT

- Install an [AWS CLI command line utility](#)
- Follow the [AWS CLI configuration guide](#) to setup your AWS credentials
- Verify your credentials setup. Start command prompt
(or, terminal on Linux/Mac), enter the following command:

```
$ aws iot list-things
```

It must execute without errors.

Setup device

- Pick one of the supported devices. We suggest to choose from [recommended devboards](#)
- Connect your device to your workstation via USB
- Complete [Quickstart Guide](#) steps 1-7 inclusive.
As a result, your device should be connected to the Internet
- Provision your device to AWS IoT:

```
mos aws-iot-setup --aws-region YOUR_AWS_REGION
```

This command performs certificate management for you, and onboards your device on a cloud. If your device has an [Atmel ECC508A](#) secure element attached, Mongoose OS automatically uses ECC508A chip for TLS and keeps your credentials securely stored.

Open AWS console, "IoT Core" service, and find your provisioned device in the list. On the picture, ESP8266 was used.

Internet button in 2 minutes

Check out a video below that shows how to implement Internet button functionality on AWS IoT in less than 2 minutes:



Controlling LED using AWS device shadow

The firmware that we've build following quickstart guide, contains code that synchronises device LED with the [desired.on](#) shadow setting.

On the AWS console, click on the device ID, choose "Shadow" tab. Click on "Edit" to modify shadow, and add the desired section:

```
{  
  "desired": {  
    "on": true  
  },  
  ...  
}
```

Click "Save". Click on "Edit" again, change on to false. Click "Save".

See device logs, and see how LED turns on and off. NOTE: on some boards, (notably, Espressif) LED pins are inverted, therefore LED turns on when on is false, and turns off when on is true.

Reporting metrics to AWS IoT

We'll report free RAM metric using MQTT API. Note that AWS IoT has versatile rules engine, which allows to handle incoming MQTT messages in a various ways: store in Dynamo DB, forward to your Lambda function, etc etc.

Open fs/init.js file in your favorite editor and copy/paste the following:

```
load('api_config.js');  
load('api_mqtt.js');  
load('api_sys.js');  
load('api_timer.js');  
  
let topic = 'metrics/' + Cfg.get('device.id');  
  
Timer.set(1000 /* milliseconds */, Timer.REPEAT, function() {  
  let msg = JSON.stringify({free: Sys.free_ram(), total: Sys.total_ram()});  
  print(topic, '->', msg);  
  MQTT.pub(topic, msg, 1);  
}, null);
```



Copy fs/init.js to the device and reboot the device:

```
mos put fs/init.js  
mos call Sys.Reboot
```

In the AWS IoT console, click on "Test". Into the "Subscription topic" field, enter metrics/# and click "Subscribe to topic". See how messages are coming.

Direct device control via AWS IoT

If you're using Bash shell, you can use this alias to talk to your device interactively via AWS IoT:

```
alias mos1="mos --cert-file $(mos config-get mqtt.ssl_cert) --key-file $(mos config-get mqtt.ssl_key) --port mqtts://$(mos  
mos1 ls -l  
init.js 330  
index.html 250  
...
```

For more in-depth explanation, see

[Secure remote device management with Mongoose OS](#)

blog post.

OTA update via AWS IoT device shadow

See [OTA via shadow](#)

Using AWS IoT in your custom firmware

The AWS IoT integration described above is implemented by the [aws](#) Mongoose OS library.

It provides device shadow API in both C/C++ and JavaScript, allowing developers to quickly prototype the device logic in JS before jumping to the C/C++ implementation.

Also, MQTT API is available, also for both C/C++ and JavaScript.

In order to have AWS IoT functionality in your custom application, make sure to add `aws` library to your `mos.yml` file:

```
libs:  
  ...  
  - origin: https://github.com/mongoose-os-libs/aws # <-- Add this!
```

Then, rebuild your app:

```
mos build
```

- [Amazon AWS IoT](#)
- [Google IoT Core](#)
- [IBM Watson](#)
- [Microsoft Azure IoT](#)
- [Private MQTT server](#)
- [Samsung Artik](#)

List all RPC services exported by device

```
mos call RPC.List  
[  
  "FS.Umount",  
  ...
```

You can see what arguments an RPC service expects by calling `RPC.Describe`, which provides a simple introspection:

```

mos call RPC.Describe '{"name": "Config.Set"}'
{
  "name": "Config.Set",
  "args_fmt": "{config: %M}"
}

```

It works because C API for registering an RPC service has JSON format specification for the input arguments. That's why Mongoose OS knows about JSON formats for all registered RPC services.

Prepare Google Spreadsheet

- Create and empty Google spreadsheet
- Click on Tools / Script Editor, paste the following code:

```

function doPost(e) {
  var doc = SpreadsheetApp.getActiveSpreadsheet();
  var sheet = doc.getSheets()[0];
  var row = JSON.parse(e.postData.contents) || [];
  row.unshift(new Date()); // Add timestamp as a first field
  var r = sheet.getRange(sheet.getLastRow() + 1, 1, 1, row.length);
  r.setValues([row]);
  return ContentService.createTextOutput('');
}

```

- Click "save". Choose a name if prompted
- In the menu, choose Publish / Deploy as web app.
Choose Project version: "New", Execute the app as: "Me", Who has access: "Everyone, even anonymous". Click deploy
- On Authorization prompt, click "Review permissions", allow this application to see, edit, create and delete spreadsheets
- Click deploy. Copy the script URL, it should be in form
<https://script.google.com/macros/s/xxxxxx/exec>

Add reporting logic to the firmware code

A firmware must send a POST request do the script URL.
A POST data must be a JSON array. Each element of the array will be inserted into a spreadsheet in a separate column. For example, if a POST data is [1234, "hello"], then in a spreadsheet a new row will be inserted with 3 columns: timestamp, 1234, and "hello".

This is a minimal app that reports a single value (free RAM) to the spreadsheet:

```

# include "mgos.h"
<p class="mume-header" id="include-mgosh"></p>

const char *s_url = "https://script.google.com/macros/s/xxxxxx/exec";

static void timer_cb(void *arg) {
    char buf[100];
    struct json_out out = JSON_OUT_BUF(buf, sizeof(buf));
    json_printf(&out, "[%d]", mgos_get_free_heap_size());
    mg_connect_http(mgos_get_mgr(), NULL, NULL, s_url, NULL, buf);
    (void) arg;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_set_timer(3000, MGOS_TIMER_REPEAT, timer_cb, NULL);
    return MGOS_APP_INIT_SUCCESS;
}

```

File system

Mongoose OS implements Virtual File System layer, VFS. That means it can attach (mount) different storage types into a single file system tree.

For example, a device can have an SPI flash storage and an SD card storage. For each storage type, a filesystem driver must be implemented. For example, it is possible to write a driver that implements a Dropbox or Google Drive storage type, and a device (e.g. ESP8266 module) can mount a Dropbox folder.

Mongoose OS provides a

[Filesystem RPC service](#)

that allows remote filesystem management - for example, you can edit files remotely.

Adding file system

Let's see how to attach an additional filesystem to Mongoose OS in practice.

ESP8266

On ESP8266, Mongoose OS uses approximately 3MB of flash and default file system is 256K.

Most modules (ESP-12F have 4MB), so the last 1MB or so is available. Let's create a 512KB SPIFFS file system in this free space.

Note: we cannot use 1MB because at the end of flash there is system params area 16K in size which is used by SDK and SPIFFS requires size to be a power of 2.

First, flash the default firmware and verify that you have at least 4M of SPI flash, watch for the boot message:

```

$ mos flash esp8266 && mos console
Fetching https://mongoose-os.com/downloads/esp8266.zip...
...
[Jul 28 13:51:11.662] esp_mgos_init2      default 1.0 (20170728-103845/???)  

[Jul 28 13:51:11.668] esp_mgos_init2      Mongoose OS 2017072810 (20170728-103845/???)  

[Jul 28 13:51:11.675] esp_mgos_init2      SDK 2.1.0(ce90efd); flash: 4M; RAM: 52184 total, 49140 free  

...

```

flash: 4M means we're good to go. There are modules with even bigger flash chips - [WEMOS D1 mini Pro](#) has 16MB, you can have extra 8MB filesystem there.

Now, create the file system:

```
$ mos call FS.Mkfs '{"dev_type": "sysflash", "fs_type": "SPIFFS", "fs_opts": "{\"addr\": 3145728, \"size\": 524288}"}
Using port /dev/ttyUSB0
null
```

addr is the offset from the beginning of the flash chip where to create the filesystem, 3145728 is just after the first 3M.

Mount the newly-created filesystem:

```
$ mos call FS.Mount '{"dev_type": "sysflash", "fs_type": "SPIFFS", "fs_opts": "{\"addr\": 3145728, \"size\": 524288}", "path": "/mnt"}'
Using port /dev/ttyUSB0
null
```

List files on the new FS - it will be empty:

```
$ mos ls -l /mnt
Using port /dev/ttyUSB0
```

Let's put a file there

```
$ mos put README.md
$ mos ls -l /mnt
Using port /dev/ttyUSB0
README.md 991
```

To make the device attach the file system automatically at boot, let's configure the sys.mount section:

```
$ mos config-set sys.mount.path=/mnt sys.mount.dev_type=sysflash sys.mount.fs_type=SPIFFS 'sys.mount.fs_opts={"addr": 3145728, "size": 524288}' &&
Using port /dev/ttyUSB0
Getting configuration...
Setting new configuration...
Saving and rebooting...
Using port /dev/ttyUSB0
...
[Jul 28 14:02:32.787] esp_mgos_init2      default 1.0 (20170728-103845/??)
[Jul 28 14:02:32.793] esp_mgos_init2      Mongoose OS 2017072810 (20170728-103845/??)
[Jul 28 14:02:32.801] esp_mgos_init2      SDK 2.1.0(ce90efd); flash: 4M; RAM: 52184 total, 49140 free
[Jul 28 14:02:32.805] esp_print_reset_info Reset cause: 4 (soft reset)
[Jul 28 14:02:32.810] mgos_vfs_dev_open   sysflash () -> 0x3fff0034
[Jul 28 14:02:32.820] mgos_vfs_mount     Mount SPIFFS @ / (dev 0x3fff0034, opts {"addr": 32768, "size": 262144}) -> 0x3fff0044
[Jul 28 14:02:32.883] mgos_vfs_mount     /: size 233681, used: 99647, free: 134034
[Jul 28 14:02:32.970] mgos_sys_config_init MAC: 1AFE34A5930F
[Jul 28 14:02:32.975] mgos_sys_config_init WDT: 30 seconds
[Jul 28 14:02:32.979] mgos_vfs_dev_open   sysflash () -> 0x3fff087c
[Jul 28 14:02:32.989] mgos_vfs_mount     Mount SPIFFS @ /mnt (dev 0x3fff087c, opts {"addr": 3145728, "size": 524288}) -> 0x3fff088c
[Jul 28 14:02:33.120] mgos_vfs_mount     /mnt: size 474641, used: 1255, free: 473386
[Jul 28 14:02:33.125] mgos_mdns_init    Listening on udp://:5353
...
```

We can make built-in HTTP server serve from /mnt instead of /:

```
$ mos config-set http.document_root=/mnt
```

ESP32

Typical ESP32 modules have 4M flash. Code size is bigger, there is approximately 400K available after all the system stuff. Let's add a 256K file system.

ESP32 uses [partition tables](#) for allocating flash space, we will need to add a partition and recompile firmware.

Check out or download the [default app](#), edit the [mos.yml](#) file and edit the esp32-specific section:

```

- when: mos.platform == "esp32"
  apply:
    build_vars:                                # Add these
      ESP_IDF_EXTRA_PARTITION: fs_ext,data,spiffs,,256K  # two lines

```

Build and flash the firmware:

```

$ mos build --verbose --arch esp32 --clean && mos flash && mos console
...
Connecting to https://mongoose.cloud, user test
Uploading sources (2878 bytes)
...
Success, built default/esp32 version 1.0 (20170728-131414/??).
Firmware saved to build/fw.zip
...
[Jul 28 14:14:44.990] I (99) boot: Partition Table:
[Jul 28 14:14:44.990] I (110) boot: ## Label      Usage      Type ST Offset Length Flags
[Jul 28 14:14:44.990] I (135) boot: 0 nvs      WiFi data   01 02 00009000 00004000 00000000
[Jul 28 14:14:45.012] I (161) boot: 1 otadata  OTA data    01 00 0000d000 00002000 00000000
[Jul 28 14:14:45.012] I (186) boot: 2 app_0    OTA app     00 10 00010000 00180000 00000000
[Jul 28 14:14:45.013] I (212) boot: 3 fs_0    SPIFFS     01 82 00190000 00040000 00000000
[Jul 28 14:14:45.034] I (238) boot: 4 app_1    OTA app     00 11 001d0000 00180000 00000000
[Jul 28 14:14:45.035] I (263) boot: 5 fs_1    SPIFFS     01 82 00350000 00040000 00000000
[Jul 28 14:14:45.052] I (289) boot: 6 fs_ext   SPIFFS     01 82 00390000 00040000 00000000
[Jul 28 14:14:45.053] I (315) boot: End of partition table
...

```

Note presence of the extra `fs_ext` partition at the end.

Create the file system:

```

$ mos call FS.Mkfs '{"dev_type": "esp32part", "dev_opts": "{\"label\": \"fs_ext\"}", "fs_type": "SPIFFS"}'
Using port /dev/ttyUSB0
null

```

There is no need to specify size and offset, they are taken from partition information.

Mount the file system:

```

$ mos call FS.Mount '{"dev_type": "esp32part", "dev_opts": "{\"label\": \"fs_ext\"}", "fs_type": "SPIFFS", "path": "/mnt"}'
Using port /dev/ttyUSB0
null

```

To make the device attach the file system automatically at boot, let's configure the `sys.mount` section:

```

$ mos config-set sys.mount.path=/mnt sys.mount.dev_type=esp32part sys.mount.fs_type=SPIFFS 'sys.mount.dev_opts={"label": "fs_ext"}'

```

CC3200

There is no space available on the CC3200 chip for extra file system, but external SPI flash can be attached - see below.

External SPI flash

Mongoose OS has an [external SPI flash driver](#), which can be used to attach.

Having attached an SPI flash chip, you will be able to use the `spi_flash` driver like so:

```

$ mos call FS.Mkfs '{"dev_type": "spi_flash", "dev_opts": "{\"freq\": 20000000, \"cs\": 0}", "fs_type": "SPIFFS", "fs_opts": "{\"size\": 1048576}"}'

```

- [List RPC services](#)
- [Get device info](#)

- [Add filesystem](#)
- [MakerFactory ESP32 board](#)
- [Add FS with content](#)
- [Send data to spreadsheet](#)

HowTo: Create an image file of an populated filesystem

When developing with **Mongoose OS** and IoT devices to flash, it is often necessary to provide data that takes up more space than the average file system can provide. There exists a nice [HowTo](#) that explains how to add an additional filesystem to a device running **Mongoose-OS** - but the problem is: it's a lot of manual effort and it's a bit complicated. To make matters worse, the file system must also be "filled" manually for each individual device. So if you plan to flash many devices, you need to find a way to create a firmware package that already contains the file system. This will then be transferred to the device in one go - there is no need to intervene manually - everything is included in one firmware.

This HowTo describes how to achieve this goal. This requires some preparatory work and a few steps, but the result is worthwhile. And it doesn't matter if it's an additional 256kB file system or one with 16MB - in both cases you can cover it with a single firmware package and save a lot of trouble.

1. Necessary Preparations

The first step is to go the usual way - set up your device the way you want it, create the additional file system and store the desired files in it. There are two ways to do this:

- on the one hand the **conventional way** to create such a file system as described in [this manual](#), or
- on the other hand the **use of the library "fstab"**, which does the work of creating and mounting the file system for you.

It is sufficient to use the first way, because you only have to set up and mount the file system once. All in all, the effort is even a little less than with the external library, so here we describe the first way. A tutorial on how to use the "fstab" library may be created elsewhere, then you can decide yourself which way you want to go.

2. Creating and Populating an Additonal Filesystem

This part is not very tiring: proceed as described under "[Adding File System](#)", set up the filesystem and mount it under the desired name, e.g. **"%mnt"** (this is our example) or another name as you like. The spelling of the filesystem reminds of a directory and is also used in the path syntax, but it is ultimately a mounting point for the files, the filesystem below is flat.

Once your device is newly flashed and the file system is active, you can test it with e.g.

```
mos[.exe] --port <portname> ls /mnt
```

If it is not available, an **error message** is displayed, if it is active, nothing is shown because it is empty - but without an error message. Very simple.

Now you have to copy the desired files into the directory - there are a few points to consider. Files will be transferred to the device with the command

```
mos[.exe] --port <portname> put <local relative path>/<filename> <file system of the device>/<filename>
```

Unfortunately `mos[.exe]` may not transfer a whole bunch of files from a folder - every file has to be copied one after another with the full command line. So if you have to transfer a lot of files, it might be helpful to write this command down to a *shell script* or *batch file*, so you may edit and adapt the commands to your need and then execute them via the *shell script/batch file* - this will be much faster than doing it by hand.

Be aware: the command to put a file to an additional file system needs the complete target filename including the mountpoint. This is mandatory If you only use the mount point - so to speak as a target folder - no error will be shown, but the data will be copied into a file with an empty filename. This is kind of a bug, but you may avoid it.

So if you have finished this step successfully, you may examine your files system again with this command:

```
mos[.exe] --port <portname> ls -l /mnt
```

and you should see something like this (of course with different port and file names, but the example is clear):

```
macuser@Mac-Pro:~$ mos --port /dev/cu.SLABS_USB-to-UART ls -l /mnt
Bulb-green.dif 3024
Bulb-off.dif 3024
Bulb-red.dif 3024
Bulb-yellow.dif 3024
Switch-off.dif 11108
Switch-on.dif 11108
Titles-Ident.dif 6046
Titles.dif 18188
config.min.css.gz 1488
config.min.js.gz 1819
index.html 4962
logo.svg.gz 1142
mainswitch-status.json 19
tz-archive.zip 5220
workplaces.min.json 417
zepto.min.js.gz 10295
macuser@Mac-Pro:~$
```

The parameter **-l** tells `mos[.exe]` to list the file sizes too - it's a helpful information. With this process you have now created the "original file system", which is now used as the basis for the image file. For this reason you should be aware that all files are available in the desired version and form (e.g. gzipped).

3. Getting the Image

Now it's time to transfer the filesystem in the device to the local images file. So there has to be some information gathering first. When you boot your device and you have the serial monitor aka console active with this command:

```
mos[.exe] --port <portname> console
```

then you should see something like this scrolling in the terminals window:

The **red marked** entries are describing the filesystem you want to create an image from:

- "*Name/Label*" is the name or label of the existing filesystem, often also called "fs_ext"
- "*Offset (HEX)*" is the start address of the file systems data
- "*Size (HEX)*" is the size or length of the whole file system (even if it is empty)

With these parameters you may create the command to get the data to your local hard drive. For this you need the Python tool **"esptool.py"** from `espressif`. Since this tool is also used by `mos[.exe]`, you will also have it on your system, but you can as well download it [from here](#). You'll find some documentation how to use the Python script too at this linked place. That Pathon must be installed is obvious, of course - this will not be further explained here.

```
esptool.py -p <portname> -b 1500000 read_flash 0x390000 0xC00000 mnt12mb.img
```

As you see, the parameters after `read_flash` are describing the start and the length of the data block to be read from the device and written to the file - these are the values we investigated before (see illustration above). In this example you see that a big file will be created - it is the additional file system of an ODROID-GO which comes with 16MB flash - as the system uses 4MB, there are still 12MB remaining for

additional file system use. But even if you have an system like e.g. an ESP32 developer bord with 4MB, you may use a space of 256kB as external file system. This already may help a lot to put some files aside which e.g. you may use for web server purposes or symbols to be displayed on an TFT or OLED display. You may put the created image file e.g. in the `src` subfolder of your application folder, it will be used later.

4. Configuring `mos.yml`

Now we have to look for some settings in `mos.yml` to integrate the newly created image file into the build package `fw.zip`.

In this file you need to have the following lines in the `build_vars` section:

```
build_vars:  
  ESP_IDF_EXTRA_PARTITION: data,data,spiffs,,12288K  
  APP_EXTRA_FW_PARTS: fs_data:type=fs,src=src/mnt12mb.img,ptn=data
```

The first line should already exist as you have created the file system according to the mentioned HowTo. Perhaps the name may differ and you used `fs_ext` instead of `data` - but this won't play a big role. As you see, you have to enter the path of the created image file under the `src`=entry, and the name of the file system under `ptn`. As mentioned, you may exchange `data` found here with `fs_ext` in your own version. But be aware that the second parameter of the entry `ESP_IDF_EXTRA_PARTITION` has allways to be `data` no matter what the name is you've chosen for your file system.

After these settings you have to tell the system to mount the file system when booting - so you have to enter the following lines (if using another platform than ESP32 as in this example you have to look which parameters are valid in the particular case), for this use, you'll have to include the library `fstab`:

```
config_schema:  
  - ["fstab.fs0.dev", "data"]  
  - ["fstab.fs0.type", "LFS"]  
  - ["fstab.fs0.opts", '{"bs": 4096}']  
  - ["fstab.fs0.path", "/mnt"]  
  - ["fstab.fs0.create", false]
```

Under `fstab.fs0.type` you have to enter which file system type is used - this depends on the created file systems in the first place. So in this example the file system type `LFS` (Little File System) is chosen for a bigger partition because it's way faster than `SPIFFS` and handles big file systems quite well. But as it wastes a lot of space, you should use `SPIFFS` for smaller filesystems. The entry `fstab.fs0.create` has to be set to `false`, if not, the system will create a new one on start - so your existing data won't be used in the file system, because the new file system will be empty.

If you want to use `LFS`, you have to include the appropriate library in the configuration file:

```
libs:  
  - origin: https://github.com/mongoose-os-libs/vfs-fs-lfs
```

Optionally - if you use the http server - you may enter also the following line under `config_schema`:

```
- ["http.document_root", "/mnt"]
```

This will use the mounted file system as DOCROOT for the web server as well.

5. Goal reached!

So this is it! With these steps taken, after the next flash build process the image file will be integrated into the `fw.zip` package and flashed to the device. No further action necessary. So using this method enables you to create firmware packages with additional file systems which might be installed also via OTA without the need of individually handling the setup up of the file system.

6. Some Remarks and Hints

This example is based on the challenge to fully use the flash memory of an [ODROID-GO](#), a kind of "GameBoy like" designed device (ESP32, 16MB Flash, 4MB PSRAM).

So why not just define the standard filesystem bigger just like shown below if you have 16MB available?

```
build_vars:  
  ESP_IDF_SDKCONFIG_OPTS: >  
    ${build_vars.ESP_IDF_SDKCONFIG_OPTS}  
  CONFIG_ESPTOOLPY_FLASHSIZE_4MB=  
  CONFIG_ESPTOOLPY_FLASHSIZE_16MB=y  
  CONFIG_ESPTOOLPY_FLASHSIZE="16MB"  
  FLASH_SIZE: 16M  
  FS_SIZE: 6291456
```

The explanation: Simply setting the value of `FLASH_SIZE` and `FS_SIZE` to higher values using all the space was not just like this possible, because it will lead to a division of the flash (because of the OTA architecture, I assume) and you can only use half of it.

Anything over **6MB** (half the space of the remaining **12MB** available after subtraction of **4MB** needed by the system from the **16MB** total) led to build process errors. In addition, the standard file system is formatted with `SPIFFS`, which leads to long processing times for large partitions, which sometimes even cause the watchdog to be activated. For this reason, it is recommended to use the file system `LFS` ("Little File System"), but this must then be set up as an additional file system.

And even if you have only a little amount of extra space, it will be better to add this as an additional file system, because if you increase the standard file system, you only may use half of the remaining space, for the reasons explained above.

7. Troubleshooting

If you have already stored a file system in the older LFS version 1, this won't work out of the box like described above. In this case you may set

```
cdefs:  
  MGOS_LFS1_COMPAT: 1
```

in the `mos.yml` file, so the driver will be compatible with the old version. But this is only a solution to perhaps convert old data, in daily use, the newer version is the better choice.

MakerFactory board + Mongoose OS

This tutorial demonstrates how to use Mongoose OS on MakerFactory ESP32 development board, and use Google IoT core as a cloud backed to control the board and report metrics.

Setup the board and Google IoT Core

Please follow [Google IoT Core tutorial](#) in order to setup the cloud side and provision your board to Google IoT Core. You need to complete **Setup Google IoT Core** and **Setup device** sections only. When done, continue on this tutorial.

NOTE: before executing `mos flash` command, press and hold `IO0` button on the board in order to enter programming mode. That needs to be done every time you flash the board.

HINT FOR THE MANUFACTURER:

In order to omit manual button press, modify USB-to-Serial adapter wiring by connecting DTR to GPIO0 and RTS to RESET. See notes from Espressif at <https://github.com/espressif/esptool/wiki/ESP8266-Boot-Mode-Selection#automatic-bootloader>

Controlling LED via Google IoT Core

Open `fs/init.js` in your favorite editor, copy-paste the following snippet and save:

```
load('api_config.js');
load('api_gpio.js');
load('api_mqtt.js');
load('api_neopixel.js');

let strip = NeoPixel.create(33, 2, NeoPixel.GRB);
strip.clear();

let topic = '/devices/' + Cfg.get('device.id') + '/config';
MQTT.sub(topic, function(conn, topic, msg) {
  print('Topic:', topic, 'message:', msg);
  // Expected config format: {"led": 0, "r": 123, "g": 123, "b":123}
  let obj = JSON.parse(msg) || {};
  strip.setPixel(0, obj.r || 0, obj.g || 0, obj.b || 0);
  strip.show();
}, null);
```

The snippet above initialises RGB LED strip, and subscribes to the `/config` object notifications. Once the config object is sent to device by Google IoT Core, the handler function is called and lights the corresponding LED.

In the command prompt (or terminal on Linux/Mac), enter the following commands to copy `init.js` to the device, reboot the device, and start monitoring serial output:

```
mos put fs/init.js
mos call Sys.Reboot
mos console
```



Find a registered device in a Google IoT Core device registry. Click on "Update Config" button, and in the opened dialog, enter the following configuration and hit "SEND TO DEVICE":

```
{"r": 50, "g": 0, "b": 0}
```

Note how one of the LEDs turns red. Enter

```
{"r": 0, "g": 0, "b": 0}
```

Note how the LED turns off.

Important thing to note: Google IoT Core send device config object each time device comes online or a config change is made. That means, that if you power cycle the device, it'll sync its LED state with the cloud as soon as it gets connected.

Reporting state to Google IoT Core

We are going to connect a Grove DHT temperature sensor to the board, then modify `fs/init.js` to periodically read the temperature and send it over to the Google IoT Core.

Get the Grove DHT sensor and connect it to the IO17 Grove connector on the board, as shown on the picture. Make sure to enable IO17 via the switch, as shown on the picture.

The DHT driver is built-in to the `demo.js` app we're using, therefore we do not need to rebuild the firmware.

Open `fs/init.js` in your favorite editor, copy-paste the following snippet and save:

```
load('api_config.js');
load('api_dht.js');
load('api_mqtt.js');
load('api_timer.js');

let topic = '/devices/' + Cfg.get('device.id') + '/state';
let pin = 17; // GPIO pin which has a DHT sensor data wire connected
let dht = DHT.create(pin, DHT.DHT11); // Initialize DHT library

Timer.set(3000 /* milliseconds */, Timer.REPEAT, function() {
  let t = dht.getTemp();
  let h = dht.getHumidity();
  if (isNaN(h) || isNaN(t)) {
    print('Failed to read data from sensor');
  } else {
    let msg = JSON.stringify({temperature: t, humidity: h});
    print(topic, '->', msg);
    MQTT.pub(topic, msg, 1);
  }
}, null);
```



In the command prompt (or terminal on Linux/Mac), enter the following commands to copy `init.js` to the device, reboot the device, and start monitoring serial output:

```
mos put fs/init.js
mos call Sys.Reboot
mos console
```

In the Google IoT Web console, click on the "Configuration and state history", and see how new state objects are arriving. Click on any state object, select "text" view, and check the JSON string that is sent by device.

Get device information

The solution is to call `Sys.GetInfo` RPC service.

From your terminal, run:

```
mos call Sys.GetInfo
[Aug 21 08:29:17.016] {
...
[Aug 21 08:29:17.016]   "mac": "5ECF7F060046",
[Aug 21 08:29:17.016]   "arch": "esp8266",
[Aug 21 08:29:17.016]   "wifi": {
[Aug 21 08:29:17.016]     "sta_ip": "192.168.1.31",
...
}
```

You can see e.g. MAC and IP addresses in the output.

In order to call the same function in JS, do:

```
load('api_rpc.js');
RPC.call(RPC.LOCAL, 'Sys.GetInfo', null, function(resp, ud) {
  print('Response:', JSON.stringify(resp));
  print('MAC address:', resp.mac);
}, null);
```

The device would print to the output:

```
[Aug 21 08:23:33.162] Response: {... "arch":"esp8266",... }
[Aug 21 08:23:33.170] MAC address: 5ECF7F060046
```

Advanced guide: C/C++

In this guide we will learn how to implement device logic in C/C++.

In particular, our firmware will:

- have an ability to read DHT22 temperature/humidity sensor data
- set up a timer to read and print sensor data to the debug console
- implement a RPC service for reading the sensor remotely - for example, via MQTT

1. Start mos tool
2. Clone template project
3. Learn app dir structure
4. Add required libraries
5. Create app's custom config
6. Add sensor code
7. Build app, test sensor
8. Add RPC service
9. Rebuild app and test RPC

Start mos tool

Run `mos` tool without arguments to start the Web UI. After start, the Web UI changes current working directory to the directory where it finishes the last time. In other words, it "remembers" its settings: the working directory, chosen port, board, etc.

In this example, it is an `app1` directory, a quickstart example I have done recently:

Since we are going to create our new app in a different directory,

use the `cd DIRECTORY` command to change the current directory.

I am going to do `cd ..` to go up one level. Notice the current directory change:

Clone template project

Now we are going to use a `mos clone URL DIRECTORY` command, in order to clone some remote app into a DIRECTORY. Press `Ctrl-n` - that populates the input field with `mos clone https://github.com/mongoose-os-apps/demo-js app1`. We don't want to use `demo-js` as a template, so change it to `empty` to use a minimal app, and change `app1` to `app2`:

Now press Enter to execute the command. Notice that the mos tool automatically changes enters into the cloned directory:

Learn app directory structure

Click on the folder icon on the bottom left corner to open a system file browser in the current directory:

Here is the meaning of all files:

```
fs/      -- All files we put here, will end up on device's filesystem
└ index.html -- Device's HTTP server, if enabled, will serve this file
LICENSE
mos.yml    -- Describes how to build an app
README.md   -- Document your app in this file
src/
└ main.c    -- Contains device logic. We are going to edit this file
```

Add necessary libraries

Open `mos.yml` file in your favorite editor and add support for DHT sensor:

```
libs:
- origin: https://github.com/mongoose-os-libs/rpc-service-config
- origin: https://github.com/mongoose-os-libs/rpc-service-fs
- origin: https://github.com/mongoose-os-libs/rpc-uart
- origin: https://github.com/mongoose-os-libs/wifi
- origin: https://github.com/mongoose-os-libs/dht # <-- Add this line!
```

Note - all available libraries are collected under the <https://github.com/mongoose-os-libs> organisation. They are categorised and documented under the "API Reference" docs section.

Now, open `src/main.c`, you'll see the following skeleton code which initialises an app that does nothing:

```
# include "mgos.h"
<p class="mume-header" id="include-mgosh-1"></p>

enum mgos_app_init_result mgos_app_init(void) {
    return MGOS_APP_INIT_SUCCESS;
}
```

Create custom configuration section

Let's add a code that reads from a DHT temperature sensor every second.

The pin to which a sensor is attached, we make configurable by editing a `config_schema:` section in the `mos.yml`, to have it like this:

```
config_schema:  
- ["app.pin", "i", 5, {title: "GPIO pin a sensor is attached to"}]
```

This custom configuration section will allow us to change sensor pin at run time, without recompiling firmware. That could be done programmatically or via the `mos` tool, e.g. `mos config-set app.pin=42`.

NOTE: see [mos.yml file format reference](#)
for the full documentation about the `mos.yml`.

Add code that uses sensor library

Then, edit `src/main.c`, add a timer (see [timer api docs](#)) that reads DHT and logs the value (error handling is intentionally omitted):

```
# include "mgos.h"  
<p class="mume-header" id="include-mgosh-2"></p>  
  
# include "mgos_dht.h"  
<p class="mume-header" id="include-mgos_dht"></p>  
  
static void timer_cb(void *dht) {  
    LOG(LL_INFO, ("Temperature: %lf", mgos_dht_get_temp(dht)));  
}  
  
enum mgos_app_init_result mgos_app_init(void) {  
    struct mgos_dht *dht = mgos_dht_create(mgos_sys_config_get_app_pin(), DHT22);  
    mgos_set_timer(1000, true, timer_cb, dht);  
    return MGOS_APP_INIT_SUCCESS;  
}
```

The `mgos_dht.h` file comes from the `dht` library that we have included to our app. In order to find out its documentation and API, navigate to "API Reference" -> "Drivers" -> "DHT temp sensor". This should bring you to [this page - DHT temp sensor](#). Similarly, you can find out about any other library.

Build the app and test sensor reading

Connect DHT sensor to pin 5. The sensor itself has following pins:

This is an example with ESP8266 NodeMCU. Red connector is VCC 3.3 volts, black connector is ground GND, and yellow is data, connected to pin 5:

Build, flash the firmware, and attach the console to see device logs. Assume we're working with ESP8266:

Choose your board and port in the the UI, and run `mos build` command:

When finished, run `mos flash` to flash the firmware and see the output in the console:

Add RPC service to read sensor data remotely

Now let's use the cornerstone of Mongoose OS remote management capabilities.

We can make any hardware function be remotely accessible. This is done by creating an RPC service. Read more about it in the Overview and Core libraries sections, and here we jump straight to it. Looking at

[MG-RPC API doc](#), add RPC service Temp.Read:

```
# include "mgos.h"
<p class="mume-header" id="include-mgosh-3"></p>

# include "mgos_dht.h"
<p class="mume-header" id="include-mgos_dht-1"></p>

# include "mgos_rpc.h"
<p class="mume-header" id="include-mgos_rpch"></p>

static void timer_cb(void *dht) {
    LOG(LL_INFO, ("Temperature: %lf", mgos_dht_get_temp(dht)));
}

static void rpc_cb(struct mg_rpc_request_info *ri, void *cb_arg,
                   struct mg_rpc_frame_info *fi, struct mg_str args) {
    mg_rpc_send_responsef(ri, "{value: %lf}", mgos_dht_get_temp(cb_arg));
    (void) fi;
    (void) args;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_dht *dht = mgos_dht_create(mgos_sys_config_get_app_pin(), DHT22);
    mgos_set_timer(1000, true, timer_cb, dht);
    mg_rpc_add_handler(mgos_rpc_get_global(), "Temp.Read", "", rpc_cb, dht);
    return MGOS_APP_INIT_SUCCESS;
}
```

Build and test the final app

Run `mos build` followed by `mos flash`.

And now, call the device's RPC service by running `mos call Temp.Read`.

You will see `{"value": 18.6}` printed.

This call could be performed over the serial connection as well as over network connection - see [RPC section](#) to learn more.

Supported hardware

Mongoose OS supports the following microcontrollers:

Vendor	Microcontrollers
STMicroelectronics	STM32L4, STM32F4, STM32F7 series
Texas Instruments	CC3220, CC3200
Espressif Systems	ESP32, ESP8266

Development boards reference

STM32 B-L475E-IOT01A

Capabilities	WiFi, BT, 868 RF, NFC, 128k RAM, 1M Flash
Flash demo-js app	<code>mos flash stm32-B-L475E-IOT01A</code>
Build an app	<code>mos build --platform stm32 --board B-L475E-IOT01A</code>

[Buy on mouser.com](#)

STM32 NUCLEO-F746ZG

Capabilities	Ethernet, 320k RAM, 1M Flash
Flash demo-js app	<code>mos flash stm32</code>
Build an app	<code>mos build --platform stm32 --board NUCLEO-F746ZG</code>

[Buy on mouser.com](#)

TI CC3220SF LaunchPad

Capabilities	WiFi, 256k RAM, 1M Flash
Flash demo-js app	<code>mos flash cc3220</code>
Build an app	<code>mos build --platform cc3220</code>

[Buy on mouser.com](#)

ESP32 PICO-D4-KIT

Capabilities	WiFi, BT, 520k RAM, 4M Flash
Flash demo-js app	<code>mos flash esp32</code>
Build custom app	<code>mos build --platform esp32</code>

[Buy on mouser.com](#)

ESP8266 NodeMCU

Capabilities	WiFi, 96k RAM, 4M Flash
Flash demo-js app	<code>mos flash esp8266</code>

Build an app

```
mos build --platform esp8266
```

[Buy on aliexpress.com](#)

ESP32 Devkit-C

Capabilities	WiFi, BT, 520k RAM, 4M Flash
Flash demo-js app	<code>mos flash esp32</code>
Build an app	<code>mos build --platform esp32</code>

[Buy on mouser.com](#)

ESP32 ODROID-GO

Capabilities	WiFi, BT, 2.4" TFT, 4MB PSRAM, 16M Flash, SD card, Battery, Speaker, 10 Buttons, Powerswitch
Flash demo-js app	First, download conf3.json from here . Then, <code>mos flash esp32</code> <code>mos put conf3.json</code>
Build an app	<code>mos build --platform esp32</code>



[Buy on hardkernel.com](#)

Mongoose OS quick start guide

A 12-minute guide to turn your device into a mobile-controllable, updatable, remotely manageable, secure, configurable product.

1. Download and install [1 min]
2. Start mos tool [0 min]
3. Install drivers [2 min]
4. Create new app [1 min]
5. Build app firmware [1 min]
6. Flash firmware [1 min]
7. Configure WiFi [0 min]
8. Register on mDash [1 min]
9. Change firmware [2 min]
10. Update over-the-air [2 min]

1. Download and install mos tool

Mongoose OS uses `mos` tool for various tasks:

building firmware, flashing firmware,

managing device, provisioning device on cloud services, and so on.

OS	Setup instructions
Windows	Create C:\mos folder. Right-click on this mos.exe link, choose "Save link as", save <code>mos.exe</code> into the C:\mos folder. Double-click on <code>mos.exe</code> to start a Web UI. If it does not start, open command prompt, enter <code>cd c:\mos</code> and then <code>mo</code>

```
s --start-webview=false
```

First, [install brew utility](#). Then execute the following to start a Web UI:

```
brew tap cesanta/mos  
brew install mos  
mos
```

MacOS

Note: Linux version starts UI in a browser window.

```
sudo add-apt-repository ppa:mongoose-os/mos  
sudo apt-get update  
sudo apt-get install mos  
mos
```

Ubuntu Linux

Note: Linux version start UI in a browser window.

```
git clone https://github.com/mongoose-os/mos  
cd mos-tool/mos/archlinux_pkgbUILD/mos-release  
makepkg  
pacman -U ./mos-*.tar.xz  
mos
```

Arch Linux

Note: starts UI in a browser window.

```
curl -fsSL https://mongoose-os.com/downloads/mos/install.sh | /bin/bash  
mos
```

Generic
MacOS/Linux

2. Start mos tool

Make sure your device is connected via the USB to your workstation.

Completing previous section should leave you with the `mos` Web UI running:

Note: if you like using command prompt / terminal instead of the UI,
you can execute `mos` commands in a command prompt instead.

The Web UI appears only when `mos` tool is started without arguments.

On Windows and Mac, closing the Web UI window also terminates the `mos` command.

On Linux, the Web UI is started in a browser, so to stop it, close the
browser window and pressing `Ctrl-C` in a terminal where `mos` is running.

3. USB-to-Serial drivers

If you have your device connected via the USB, the UI can still
complain about the unavailable ports like this:

The most usual reason for this is a missing USB-to-Serial driver. Below are the links to the driver installation pages. Install the driver, and re-insert your device into the USB.

- [Silabs drivers](#) for Espressif boards
- [CH43x drivers](#) for Espressif boards
- [FTDI drivers](#) for CC3200, CC3220

In a "Choose port" dropdown, at least one port must appear.

NOTE: on some Mac computers, the port might not show up even after the driver is installed. The possible reason could be the extension policy, the fix is covered in [this issue](#)

4. Create new app

In the UI, select your board from the dropdown menu. Then, press `Ctrl-n` to create a new app. That will paste the following command:

```
mos clone https://github.com/mongoose-os-apps/demo-js app1
```

Press Enter. That is going to clone `demo-js` repo into the `app1` directory.

Notice how current directory changes to `app1`:

5. Build app firmware

Make sure the board is selected. Type `mos build` and press Enter.

That starts a build process that takes up to a minute to complete.

When the command finishes, the built firmware will be located at `build/fw.zip`.

As you can see, we did not make any changes to the app - we'll do that later.

6. Flash firmware

Make sure the port is selected in the dropdown menu.

Type `mos flash` and press enter. That will flash a built firmware to the device:

When flashing completes, the firmware starts to write logs to the serial console. Notice how new messages start to appear in the right window.

The window autoscrolls to the bottom, unless you scroll up manually to see some previous logs.

7. Configure WiFi

Type command `mos wifi WIFI_NETWORK_NAME WIFI_PASSWORD`. Use your own network name/password. Press enter. On the serial console, you'll see how the device reboots after saving new configuration settings, joins the WiFi network, gets the IP configuration, and synchronises time with SNTP server:

8. Add device to the mDash management dashboard

- Login to the mDash at <https://mdash.net/>.
- Click on "Add device" button. That'll create a new device.
- Click on a gears icon to trigger device management dialog.
- Click on a "Token" link to copy the device's access token to the clipboard.
- Go back to the mos tool, type command (change TOKEN to your copied token) and press enter:

```
mos config-set dash.enable=true dash.token=TOKEN
```

A device should become online ("green") on a dashboard:

Click on a gears icons to see a management dialog for a device:
 general information, configuration editor, file
 editor, device shadow editor, and an RPC service call window. We won't study
 all these management tools now though, so proceed to the next step.

9. Make changes to the firmware

Click on the folder icon on the bottom left corner. That opens a system
 file browser in the current app directory. Open `fs/init.js` in your
 favorite editor:

Paste this code snippet, that sends an MQTT message every second:

```
load('api_timer.js');
load('api_mqtt.js');

Timer.set(1000, Timer.REPEAT, function() {
    let ok = MQTT.pub('test/topic', 'hello from an updated firmware');
    print('mqtt message sent?', ok);
}, null);
```

Then, rebuild the firmware with `mos build`.

10. Update firmware over-the-air

Now let's update our device with the new firmware over the air.

Go to the mDash, click on the firmware update icon:

A file dialog should appear. In the file dialog, navigate to the
`app1/build` directory, and choose `fw.zip` file:

Notice how the progress bar appears showing the OTA progress:

During the OTA process, the device receives chunks of new firmware
 from the cloud, and serial console displays that:

Eventually, the device reboots with the new firmware and we could
 see the changed message in the serial console:

Next steps

It's time to move on to more advanced topics - learn how to add support

to different hardware peripherals, write logic in both C/C++ and JS, use API reference, send data to the databases/analytics engines like Firebase. Proceed to the [advanced guide](#).

Mongoose OS IDE

Mongoose OS provides an experimental Visual Studio Code plugin, that integrates Mongoose OS support into the VSCode.

To install it, run VSCode, search and install a "Mongoose OS IDE" extension:

Code completion

There is C/C++ and JavaScript completion support that makes writing Mongoose OS code a better experience.

The completions are made by scanning Mongoose OS core code, and all contributed libraries.

Example apps

GitHub repo	Description	Author
LilyGo-HiGrow-Sensor-v1	LilyGo HiGrow ESP32 Plant Sensor v1	Jason Harrell info@latitude17.io
alexa-microwave	Control a microwave with ESP32 via MQTT / Amazon Alexa	Steve Kasuya
aws-iot-button	Internet button on AWS IoT	mongoose-os
aws-iot-heater	Smart heater on AWS IoT	mongoose-os
aws-pico	ESP32-PICO-KIT breakout board demo for AWS IoT	Angus Gratton
aws-uart	Reading UART0 and sending the data to AWS IoT	bravokeyl bravokeyl@gmail.com
blynk	Using Blynk mobile app with Mongoose OS	mongoose-os
blynk-bme280-js	Using BME280 sensor with Blynk mobile app	Rumen Nikiforov
captive-portal	Captive portal example	mongoose-os
demo-c	A demo Mongoose OS firmware for C/C++	mongoose-os
demo-js	A JS-enabled demo Mongoose OS firmware	mongoose-os
door-bell	IoT doorbell that works via MQTT - generic MQTT server, AWS IoT, Google IoT, etc	Steve Kasuya steve.kasuya@gmail.com
door-sensor	IoT Door Sensor using a MC-38 (JavaScript)	Ben Rockwood benr@cuddletech.com
empty	A Mongoose OS app skeleton	mongoose-os
environmental-monitor	Environmental Monitor using a BME280 Sensor & SSD1306 OLED	Ben Rockwood benr@cuddletech.com
example-arduino-adafruit-ads1x15-js	Adafruit ADS1X15 usage example (JavaScript)	djzxz@wp.pl
example-arduino-adafruit-bme280-c	BME280 sensor usage example (C/C++)	mongoose-os

example-arduino-adafruit-bme280-js	BME280 sensor usage example (JavaScript)	mongoose-os
example-arduino-adafruit-ssd1306-c	Adafruit SSD1306 usage example (C/C++)	mongoose-os
example-arduino-adafruit-ssd1306-js	Adafruit SSD1306 usage example (JavaScript)	mongoose-os
example-arduino-dallas-temperature-c	Dallas Temperature sensor usage example (C/C++)	mongoose-os
example-arduino-dallas-temperature-js	DallasTemperature sensor usage example (JavaScript)	mongoose-os
example-arduino-ds3231-js	DS3231 usage example (JavaScript)	djzxz@wp.pl
example-arduino-hello-c	Arduino C/C++ basic API usage example	mongoose-os
example-arduino-onewire-c	Arduino OneWire C app	mongoose-os
example-arduino-onewire-js	Arduino OneWire API usage example (JavaScript)	mongoose-os
example-arduino-wire-c	Arduino Wire API usage example on Mongoose OS (C/C++)	mongoose-os
example-binary-lib-c	An example demonstrating usage of binary libs (.a)	mongoose-os
example-dht-c	DHT sensor usage example (C/C++)	mongoose-os
example-dht-js	DHT sensor usage example (JavaScript)	mongoose-os
example-dns-sd	DNS-SD usage example in C/C++	mongoose-os
example-m5stack-js	A JS-enabled demo application for M5stack	mongoose-os
example-neopixel-c	Neopixel strip control example	mongoose-os
example-neopixel-js	Neopixel strip control example	mongoose-os
example-no-libs-c	Most minimal app - no libs, no networking, no RPC, nothing	mongoose-os
example-rpc-c	Custom RPC service example in C/C++	mongoose-os
example-shadow-js	A device shadow example in JavaScript	mongoose-os
example-spi-c	SPI usage example in C/C++	mongoose-os
example-uart-c	UART usage example in C/C++	mongoose-os
example-uart-js	UART JavaScript API usage example	mongoose-os
example-wifi-scan-js	Wifi JavaScript API usage example	mongoose-os
http-fetch	Download URL and write content to the UART or file	mongoose-os
huzzah-featherwing	A Mongoose-OS Huzzah TFT Featherwing demo	Pim van Pelt pim@ipng.nl
lilygo-higrow-soil-temp-humi-sensor	HiGrow Plant Sensor	Luca Fabbri luca@higrow.tech
losant-motion-sensor	Losant - detecting Motion Using a PIR sensor	Taron Foxworth
losant-mqtt	Losant cloud service integration	Taron Foxworth
losant-temp-sensor	App to send ESP32/8266 temperature readings to Losant periodically.	Utkarsh Verma (utkarshverma@protonmail.com)
mq135	Using MQ135 gas sensor with Mongoose OS	Moncef AOUDIA (mf.aoudia@gmail.com)
neopixel-aws-iot	Control Neopixels from AWS IoT and an Android Companion App	anelson (austin@mgenio.com)

ota-aws-shadow	An empty app that does nothing	mongoose-os
smallest-wifi-button	The Smallest WiFi Button in the World (ESP8266, MQTT, IFTTT)	bitluni
smart-pool-pump	Smart Pool Timer Controller, integrates AWS shadow, BMP180 and ADE7912	Chizuruoke Chikwendu only1chi@gmail.com
smart-washing-machine	Making washing machine smart with WEMOS D1 mini	bitluni
smoke-detector	Smoke Detector (ESP8266, MQTT, IFTTT)	bitluni
sonoff-basic-openhab	Sonoff Basic firmware to work with openHAB	Michael Fung hkuser2001@gmail.com
wifi-scan	Scan WiFi networks on button press	mongoose-os
wifi-setup-web	Setup WiFi via Web UI	mongoose-os

- [Quick Start](#)
- [Develop in C/C++](#)
- [Develop in JavaScript](#)
- [IDE support](#)
- [Supported hardware](#)

Advanced Guide - JS

In this section we are going to implement the same functionality as in the previous section, but using JavaScript instead of C/C++. Also, we learn how to export custom C/C++ functions into JavaScript.

Please refer to the previous section on how to connect the DHT sensor.

Clone template project

To clone the template project, repeat the same steps as in the previous section:

```
git clone https://github.com/mongoose-os-apps/empty my-app
Cloning into 'my-app'...
remote: Counting objects: 68, done.
remote: Total 68 (delta 0), reused 0 (delta 0), pack-reused 68
Unpacking objects: 100% (68/68), done.
cd my-app
```

Add necessary libraries

In the previous step, we added DHT library. Do that, and also add mJS library to enable JavaScript support. Open `mos.yml` file in your favorite editor and add support for DHT sensor and JavaScript:

```
libs:
  - origin: https://github.com/mongoose-os-libs/rpc-service-config
  - origin: https://github.com/mongoose-os-libs/rpc-service-fs
  - origin: https://github.com/mongoose-os-libs/rpc-uart
  - origin: https://github.com/mongoose-os-libs/wifi
  - origin: https://github.com/mongoose-os-libs/dht # <-- Add this line!
  - origin: https://github.com/mongoose-os-libs/mjs # <-- Add this line!
```

Create custom configuration section

Same as in the previous section, add custom `app.pin` config setting by editing `mos.yml`:

```
config_schema:  
- ["app", "o", {title: "My app custom settings"}]  
- ["app.pin", "i", 5, {title: "GPIO pin a sensor is attached to"}]
```

Write device logic

Create `fs/init.js` file with the following content

(see [MG-RPC JS API reference](#)):

```
load('api_config.js');  
load('api_rpc.js');  
load('api_dht.js');  
load('api_timer.js');  
  
let pin = Cfg.get('app.pin');  
let dht = DHT.create(pin, DHT.DHT22);  
  
Timer.set(1000, true, function() {  
  print('Temperature:', dht.getTemp());  
}, null);  
  
RPC.addHandler('Temp.Read', function(args) {  
  return { value: dht.getTemp() };  
});
```

Tip: keep variable and function declaration names under 5 characters or less to minimise JavaScript memory usage.

mJS uses `double` type to hold values. Valid double hold numbers, and `NaN` values hold other types. 2 bytes out of 8 are used for type indication, and the rest

6 bytes are used for payload which depends on type.

Short strings that are 5 chars or less + terminating `\0` are stored directly in the `NaN` payload.

Longer strings are stored in a special buffer, and `NaN` payload stores a pointer to that buffer. Therefore, strings of length 5 and less take no extra memory in mJS.

Build and test the final app

Build and flash the app:

```
mos build --arch esp8266 && mos flash
```

And now, call the device's RPC service:

```
mos call Temp.Read  
{"value": 18.6}
```

Export custom function

[mJS JavaScript engine](#), which is used by Mongoose OS, allows to export C/C++ functions directly into JavaScript

via the FFI (foreign function interface) mechanism. Let's show how it is done.

Edit `src/main.c` file and add `my_func()` function there:

```
int my_func(int a, int b) {
    return a + b;
}
```

Note: if you want to export a C++ function, make sure it is declared as `extern "C"`. Add the following snippet in `fs/init.js`:

```
let f = ffi('int my_func(int, int)');
print('Calling C my_func:', f(1,2));
```

Rebuild, flash, test the app:

```
mos build --arch esp8266 && mos flash && mos console
[Aug  9 09:02:05.399] mgos_init           Init done, RAM: 52680 total, 44404 free, 44288 min free
[Aug  9 09:02:05.753] Calling C my_func: 3
```

How FFI calls are implemented

This is how it is done:

- During the build phase, Mongoose OS toolchain scans all JavaScript files present in the project.
- All `ffi(...)` JavaScript calls are collected in one list, and `build/gen/ffi_exports.c` file is generated with the signatures of C functions that JavaScript code intends to call.
- A `mgos_dlsym()` function is auto-generated that returns the address of the C function by its name, which makes `ffi()` call possible.

The `mos build` command has created `build` directory. Open

`build/gen/ffi_exports.c` file and notice a line with `my_func`:

```
const struct mgos_ffi_export ffi_exports[] = {
...
{"my_func", my_func},
...
}
```

To summarise, this section demonstrates how to use JavaScript API, libraries, and how to export custom C functions.

- [Quick Start](#)
- [Cloud Tutorials](#)
- [User Guide](#)
- [API Reference](#)
- [HOWTOs](#)

Debugging tools

Mongoose OS provides several ways to debug firmware functionality. They are listed below in the order of complexity.

Increase debug log level

There is a top-level `debug` section in the configuration. It could be inspected by clicking on a "Device config" tab in the GUI, or through the command line:

```
mos config-get debug
{
    "factory_reset_gpio": -1,
    "level": 2,
    "mbedtls_level": 0,
    "mg_mgr_hexdump_file": "",
    "stderr_topic": "esp8266_067082/log",
    "stderr_uart": 0,
    "stdout_topic": "esp8266_067082/log",
    "stdout_uart": 0,
    "udp_log_addr": ""
}
```

The default level is 2. Level -1 means no log. Level 0 is `ERROR`, 1 is `WARN`, 2 is `INFO`, 3 is `DEBUG`, 4 is `VERBOSE_DEBUG`. Level 4 means log everything possible, it is very verbose. Start with increasing to level 3.

Increasing the debug level from the command line

```
mos config-set debug.level=3
Getting configuration...
Setting new configuration...
Saving and rebooting...
```

Debug configuration settings quick reference

```
{
    "factory_reset_gpio": -1,           // Used to reset the device. Hold this pin down and reboot to reset
    "filter": "mg_,*mqtt*",            // Log filter. Prefix match glob for source file or function name
    "level": 2,                      // Log verbosity level
    "mbedtls_level": 0,                // Log verbosity level for TLS library
    "mg_mgr_hexdump_file": "",         // Set to "-" to hexdump network packets
    "stderr_topic": "",                // If set, an MQTT topic to send stderr logs to
    "stderr_uart": 0,                  // UART number for stderr logs
    "stdout_topic": "",                // If set, an MQTT topic to send stdout logs to
    "stdout_uart": 0,                  // UART number for stdout logs
    "udp_log_addr": ""                // Set to "IP_ADDR:PORT" to send logs to this UDP address
}
```

Notes:

- `mos` tool starts a UDP listener on port 1993. Therefore, if you set `debug.udp_log_addr` to `YOUR_MACHINE_IP:1993`, `mos` tool will start showing your device logs.
- If you connect to your device using via Websocket, as `ws://IP_ADDR/rpc`, then `mos` tool automatically sets the `debug.udp_log_addr` to the correct value and you don't need to set it up manually.

Analysing core dumps

If your app crashes showing a core dump message, you can start GDB against that coredump and analyse the backtrace at the time of the crash, as well as the state of the firmware.

First, save the coredump data to the `console.log` file.

Then clone the `mongoose-os` repo:

```
git clone https://github.com/cesanta/mongoose-os
```

Then from your apps' directory, do

```
make -C PATH_TO_MONGOOSE_OS_REPO/fw/platforms/esp8266 debug_coredump \
CONSOLE_LOG=$(pwd)/console.log ELF_FILE=$(ls $PWD/build/objs/fw.elf)
```

Running a heap log analyser

Heap log analyser is a tool developed by the Mongoose OS team. It is used for detecting memory leaks, and does it in a visual way, allowing to quickly pinpoint the place where a leak is happening.

ESP8266 is the only device this is supported on.

In order to use the heap log viewer, follow the steps below:

Enable heap log tracing

Edit your app's `mos.yml`, add the following section:

```
build_vars:
  MGOS_ENABLE_HEAP_LOG: 1
  MGOS_ENABLE_CALL_TRACE: 1
```

Rebuild your app:

```
mos build --clean --arch esp8266
...
```

Run your app to collect the heap log

Your console log will become very noisy lots of `hl{...}` lines.

Run your app, let it run a few cycles, and save the log somewhere.

If you're on Mac/linux,

```
mos console | tee /tmp/console.log | grep -av 'hl{'
```

This way you don't see the noise, but it's all logged in `/tmp/console.log`.

Analyse logs

Clone `mongoose-os` repo and start heap log viewer tool:

```
git clone https://github.com/cesanta/mongoose-os
cd mongoose-os/tools/heaplog_viewer/heaplog_server
go build
./heaplog_server --logtostderr --document_root .. \
--binary=YOUR_APP_PATH/build/objs/fw.elf --console_log /tmp/console.log
```

Point your browser at <http://localhost:8910/> and press "connect".

This should give you a pretty picture like this:

API Reference

Repo	Description	Author
adc	ADC support	mongoose-os
arduino-adafruit-ads1x15	Arduino Adafruit ADS1015 library	
djzxz		
arduino-adafruit-bme280	Arduino Adafruit BME280 and BMP280 library	mongoose-os
arduino-adafruit-gfx	Arduino Adafruit GFX library	mongoose-os
arduino-adafruit-htu21df	Arduino Adafruit HTU21D-F library converted for Mongoose OS	pedalPusher68
arduino-adafruit-pwm-servo	Arduino Adafruit PWM servo library	
djzxz		
arduino-adafruit(ssd)1306	Arduino Adafruit SSD1306 library	mongoose-os
arduino-adafruit-tsl2561	Arduino Adafruit TSL2561 library converted for Mongoose OS	pedalPusher68
arduino-compat	Arduino compatibility	mongoose-os
arduino-dallas-temperature	Arduino DallasTemperature library	mongoose-os
arduino-ds3231	Arduino DS3231 library	
djzxz		
arduino-md-parola	Parola text effects library for MAX 7219	mongoose-os
arduino-onewire	Arduino OneWire library	mongoose-os
arduino-pololu-vl53l0x	Arduino pololu VL53L0X library	dcato daisuke.cato@gmail.com
arduino-spi	Arduino SPI library	mongoose-os
arduino-wire	Arduino Wire library	mongoose-os
atca	ATCA support	mongoose-os
aws	Amazon AWS IoT integration	mongoose-os
azure	Azure IoT Hub integration	mongoose-os
barometer	Library that implements various barometers	Pim van Pelt pim@google.com
bh1730	BH1730 sensor support	Angus Gratton
blynk	Blynk mobile app integration	mongoose-os
bm222	A Bosch BM222 accelerometer library	mongoose-os
bme280	Moongose OS BME280/BMP280 library	Liviu Nicolescu
bt-common	Bluetooth support	mongoose-os
bt-service-config	Configuration over Bluetooth GATT service support	mongoose-os
bt-service-debug	Configuration over Bluetooth GATT service support	mongoose-os
bt-service-time	Bluetooth Current Time Service implementation	mongoose-os

ca-bundle	CA Bundle	mongoose-os
ccs811-i2c	Mongoose OS library implementing an I2C driver for CCS811	Pim van Pelt pim@google.com
core	Core meta-lib	mongoose-os
cron	A Cron lib	mongoose-os
crontab	Crontab	mongoose-os
dallas-rmt	Implements Mongoose dallas rmt driver	nliviu
dash	Support library for the device management dashboard	mongoose-os
demo-bundle	Collection of libraries for demos	mongoose-os
dht	A DHT lib	mongoose-os
dns-sd	Implements DNS-SD protocol	mongoose-os
ds3231	DS3231 I2C library	Liviu Nicolescu nliviu@gmail.com
empty	An empty lib that does nothing	mongoose-os
esp32-touchpad	ESP32 touch sensor support	mongoose-os
ethernet	Ethernet support	mongoose-os
file-logger	File logger library	mongoose-os
gcp	Google IoT Core integration	mongoose-os
http-server	HTTP server for Mongoose OS	mongoose-os
htu21df-i2c	Mongoose OS library implementing an I2C driver for HTU21D(F)	Pim van Pelt pim@google.com
i2c	I2C support for Mongoose OS	mongoose-os
ili9341-spi	Mongoose-OS library for ILI9341	Pim van Pelt pim@google.com
imu	Library that implements various IMU devices	Pim van Pelt pim@ipng.nl
ir	IR protocol library	Vladimir Dronnikov dronnikov@gmail.com
js-demo-bundle	Collection of libraries for JavaScript demos	mongoose-os
jstore	JSON Store	mongoose-os
location	Location library	mongoose-os
mcp9808-i2c	Mongoose OS library implementing an I2C driver for MCP9808	Pim van Pelt pim@google.com
md-max72xx	Port of md-max72xx as a mongoose-os lib	mongoose-os
mjs	JavaScript engine	mongoose-os
mongoose	Mongoose Embedded Web Server Library	mongoose-os
mq135	MQ135 gas sensor support	mf.aoudia@gmail.com
mqtt	MQTT protocol support	mongoose-os

neopixel	Neopixel driver	mongoose-os
onewire	Implements 1-Wire API on Mongoose OS	mongoose-os
onewire-rmt	Implements Mongoose rmt onewire driver	nliviu
ota-http-client	Implements Mongoose OS OTA HTTP client	mongoose-os
ota-http-server	Implements Mongoose OS OTA HTTP server	mongoose-os
ota-shadow	OTA firmware update via the AWS IoT device shadow	mongoose-os
pppos	PPPoS support	mongoose-os
prometheus-metrics	Implements Mongoose Prometheus Metrics endpoint	Pim van Pelt pim@google.com
pwm	PWM support	mongoose-os
rpc-azure	RPC support for Azure cloud	mongoose-os
rpc-common	Implements Mongoose OS RPC over MQTT protocol	mongoose-os
rpc-gatts	RPC over Bluetooth GATT (server)	mongoose-os
rpc-loopback	Allows to call local RPC services - adds RPC Loopback channel	mongoose-os
rpc-mqtt	Implements Mongoose OS RPC over MQTT protocol	mongoose-os
rpc-service-atca	RPC ATCA Service support	mongoose-os
rpc-service-config	RPC Config Service support	mongoose-os
rpc-service-cron	RPC Service for Crontab	mongoose-os
rpc-service-fs	RPC Filesystem Service support	mongoose-os
rpc-service-gattc	RPC GATT Client Service	mongoose-os
rpc-service-gpio	RPC GPIO Service	mongoose-os
rpc-service-i2c	RPC I2C Service support	mongoose-os
rpc-service-ota	Support for Over-The-Air update via RPC	mongoose-os
rpc-service-wifi	RPC Wifi Service support	mongoose-os
rpc-uart	Implements Mongoose OS RPC over UART	mongoose-os
sensor-utils	A Mongoose OS lib of several formulas useful for developing sensor-based apps (temperature, pressure, etc.)	Bradley Smith bradley.1.smith@gmail.com , pedalPusher68
shadow	A cross-cloud device shadow API	mongoose-os
sht31-i2c	Mongoose OS library implementing an I2C driver for SHT31	Pim van Pelt pim@google.com
si7005	Si7005 Temperature and Humidity Sensor Driver	mongoose-os
si7021-i2c	Mongoose OS library implementing an I2C driver for Si7021	Pim van Pelt pim@google.com
sntp	A SNTP library	mongoose-os
spi	Implements SPI API on Mongoose OS	mongoose-os

ssd1306	Implements SSD1306 OLED driver on Mongoose OS	brandond
stmpe610-spi	STMPE610 SPI based touch screen	Pim van Pelt pim@google.com
tmp006	A Texas Instruments TMP006 temperature sensor	mongoose-os
veml6075-i2c	Mongoose OS library for VEML6075	Pim van Pelt pim@google.com
vfs-common	VFS common bits	mongoose-os
vfs-dev-ram	Implements RAM-based VFS device	mongoose-os
vfs-dev-spi-flash	VFS device driver for SPI NOR flash	mongoose-os
vfs-fs-spiffs	Implements the mos VFS interface for SPIFFS	mongoose-os
watson	IBM Watson IoT Platform support	mongoose-os
wifi	WiFi support	mongoose-os

Configuration infrastructure

Mongoose OS uses a structured, multi-layer configuration.

It consists of two parts: a compile time part that defines configuration, and a run time part that uses configuration.

Compile-time generation

- Any piece of code that requires config setting, can define a .yaml file that describes configuration parameters for that piece of code
- User code could define it's own set of configuration parameters in it's own .yaml file
- All these yaml files are merged together during firmware compilation, and a single `sys_config_defaults.json` file is generated
- User-defined YAML file is applied last, therefore it can override any default settings specified in the system .yaml files
- Generated `sys_config_defaults.json` file represents all possible configurable settings for the firmware
- A C header and source files are also generated. C header contains a structure that mirrors `sys_config_defaults.json` file, and an API for getting and setting individual configuration values

Run-time init

- `conf0.json` - configuration defaults. This is a copy of the generated `sys_config_defaults.json`. It is loaded first and must exist on the file system. All other layers are optional.
- `conf1.json` - `conf8.json` - these layers are loaded one after another, each successive layer can override the previous one (provided `conf_acl` of the previous layer allows it). These layers can be used for vendor configuration overrides.
- `conf9.json` is the user configuration file. Applied last, on top of all other layers. `mos config-set` and `save_cfg()` API function modify `conf9.json`.

Therefore here are the rules of thumb:

- If you need to define your own config parameters, add a `config_schema`

section in the `mos.yml` file, as described in the [quick start guide](#)

- If you want to override some system default setting, for example a default UART speed, also use `config_schema` and add overrides there, see [example](#)
- If you want to put some unique information on each firmware, for example a unique ID, and optionally protect it from further modification, use any of the layers 1 through 8, e.g. `conf5.json`.
- `conf9.json` should never be included in the firmware, or it will override user's settings during OTA.

So, firmware configuration is defined by a set of YAML description files, which get translated into an opaque C structure `mgos_sys_config` and public accessors during firmware build: getters like `mgos_sys_config_get_....()` and setters like `mgos_sys_config_set_....(value)`. C code can access configuration parameters by invoking those accessors. Fields can be integer, boolean or string. C functions to retrieve and save that global configuration object are generated.

Example on how to access configuration parameters:

```
printf("My device ID is: %d\n", mgos_sys_config_get_device_id()); // Get config param
```

Example on how to set configuration parameter and save the configuration:

```
mgos_sys_config_set_debug_level(2); // Set numeric value
mgos_sys_config_set_device_password("big secret"); // Set string value
char *err = NULL;
save_cfg(&mgos_sys_config, &err); /* Writes conf9.json */
printf("Saving configuration: %s\n", err ? err : "no error");
free(err);
```

The generation mechanism not only gives a handy C API, but also guarantees that if the C code accesses some parameter, it is indeed in the description file and thus is meant to be in the firmware. That protects from the common problems when the configuration is refactored/changed, but C code left intact.

Mongoose OS configuration is extensible, i.e. it is possible to add your own configuration parameters, which might be either simple, or complex (nested).

At run time, a configuration is backed by several files on a filesystem. It has multiple layers: defaults (0), vendor overrides (1-8), and user settings (9). Vendor layers can "lock" certain parts of configuration for the user layer, and allow only certain fields to be changed. For example, end-user might change the WiFi settings, but cannot change the address of the cloud backend.

Compile time generation deep dive

Configuration is defined by several YAML files in the Mongoose OS source repository. Each Mongoose OS module, for example, crypto chip support module, can define its own section in the configuration. Here are few examples:

- `mgos_sys_config.yaml` is core module, defines debug settings, etc
- `mgos_atca_config.yaml` is a crypto chip support module
- `mgos_mqtt_config.yaml` has default MQTT server settings

As has been mentioned in the overview, you can define your own sections in the config, or override existing default values. This is done by placing a

config schema descriptor into `mos.yml`, like this:

```
config_schema:  
  - ["hello", "o", {"title": "Hello app settings"}]  
  - ["hello.who", "s", "world", {"title": "Who to say hello to"}]
```

When the firmware is built, all these YAML files get merged into one.

User-specified YAML file goes last, therefore it can override any other.

Then, merged YAML file gets translated into two C files, `mgos_config.h` and `mgos_config.c`. You can find these generated files in the `YOUR_FIRMWARE_DIR/build/gen/` directory after you build your firmware.

Here's a translation example of a custom `src/conf_schema.yaml`:

```
[  
  ["hello", "o", {"title": "Hello app settings"}],  
  ["hello.who", "s", "world", {"title": "Who to say hello to"}]  
]
```

It gets translated into the following getter and setter:

```
const char *mgos_sys_config_get_hello_who(void);  
void mgos_sys_config_set_hello_who(const char *val);
```

Then, C firmware code can access that custom configuration value:

```
printf("Hello, %s!\n", mgos_sys_config_get_hello_who());
```

Numbers are represented by integers in C, as are booleans.

Strings will be allocated on the heap.

IMPORTANT NOTE: Empty strings will be represented as `NULL` pointers, be careful.

Currently, all substructs are actually public and can be retrieved with their own getter; thus the header contains the struct definition and the getter:

```
struct mgos_config_hello {  
    char *who;  
};  
  
const struct mgos_config_hello *mgos_sys_config_get_hello(void);
```

It's useful to have universal functions which take the whole struct as a parameter. In the future though there will be an option to make some particular struct public, and by default all structs will be private.

Run time - factory, vendor, user layers

Device configuration is stored on the filesystem in several files:

- `conf0.json` - factory defaults layer
- `conf1.json` to `conf8.json` - vendor layers
- `conf9.json` - user layer

When Mongoose OS boots, it reads those files in exactly that order, merges into one, and initializes in-memory C configuration structure reflects that on-flash configuration. So, at boot time, `struct mgos_config` is initialised in the following order:

- First, the struct is zeroed.
- Second, defaults from `conf0.json` are applied.
- Third, *vendor configuration layers* 1 through 8 are loaded one after another.
- The *user configuration file*, `conf9.json`, is applied on as the last step.

The result is the state of the global struct `mgos_config`.

Each step (layer) can override some, all or none of the values.

Defaults must be loaded and it is an error if the file does not exist at the time of boot. But, vendor and user layers are optional.

Note that a vendor configuration layer is not present by default.

It is to facilitate post-production configuration: devices can be customised by uploading a single file (e.g. via HTTP POST to `/upload`) instead of performing a full reflash.

Vendor configuration is not reset by the "factory reset", whether via GPIO or web.

Field access control

Some settings in the configuration may be sensitive and the vendor may, while providing a way for user to change settings, restrict certain fields or (better) specify which fields can be changed by the user.

To facilitate that, the configuration system contains field access control, configured by the **field access control list** (ACL).

- ACL is a comma-delimited list of entries which are applied to full field names when loading config files at boot time.
- ACL entries are matched in order and, search terminates when a match is found.
- ACL entry is a pattern, where `*` serves as a wildcard.
- ACL entry can start with `+` or `-`, specifying whether to allow or deny change to the field if the entry matches. `+` is implied but can be used for clarity.
- The default value of the ACL is `*`, meaning changing any field is allowed.

ACL is contained in the configuration itself - it's the top-level `conf_acl` field. The slight twist is that during loading, the setting of the *previous* layer is in effect: when loading user settings, `conf_acl` from vendor settings is consulted, and for vendor settings the `conf_acl` value from the defaults is used.

For example, to restrict users to only being able change WiFi and debug level settings, `"conf_acl": "wifi.*,debug.level"` should be set in `conf{1-8}.json`.

Negative entries allow for default-allow behaviour:

`"conf_acl": "-debug.*,*"` allows changing all fields except anything under `debug`.

Resetting to factory defaults

If configured by `debug.factory_reset_gpio`, holding the specified pin low during boot will wipe out user settings (`conf9.json`).

Note, vendor settings, if present, are not reset.

Introduction

This overview is about the design of

[Mongoose OS](#), a firmware development framework for connected products. If you are an IoT firmware developer, Mongoose OS is for you.

Here we share our vision and the rationale for the design decisions we made. The vast majority of these decisions were driven by our work for our customers, when we developed device firmware to bring their IoT products to the market.

We noticed the following:

1. Each project was a start from scratch, more or less
2. Only ~10-20% of the time was spent on the actual product-specific logic.
The rest was a generic infrastructure - like configuration, OTA, etc

We refactored those generic pieces that take up to 90% of firmware development time, into a reusable set of components. We made it platform-independent - for example, toggling a GPIO code on Mongoose OS looks the same on all hardware platforms.

The result we called **Mongoose OS**.

Where does the name Mongoose come from?

We are targeting IoT products, where networking is crucial. We use a mature and trusted

[Mongoose Networking Library](#) as

the networking core - that is the origin of the name. The networking library uses `mg_` prefix for all API functions, and similarly Mongoose OS uses `mgos_` prefix.

Our goal is to share our experience in the hope that it'll help other developers to save a great deal of time and effort, reusing a solid and reliable basis for their products.

Architecture

Components

Mongoose OS is a framework for building apps (firmwares) for low-power microcontrollers (uC), and consists of the following main components:

- A `mos` tool. Provides device management and firmware building capabilities
- A build toolchain. This is a docker image which contains hardware vendor's SDK together with mongoose-os sources. A `mos build` command builds a firmware (we call it an "app") by taking `mos.yml` file in the current directory and invoking a build docker image either remotely (`mos build`) or locally (`mos build --local`).
- A collection of ready-to-use **apps** and **libraries**

Mongoose OS is based on the vendor's SDK, it extends the capabilities of the native SDK. For example, on ESP32 uC, Mongoose OS uses an ESP-IDF SDK, therefore it provides all capabilities that ESP-IDF provides, plus extra that come with Mongoose OS. If user code uses crossplatform API only, it can be built on all supported hardware platforms with no code changes:

If we zoom in the yellow "Mongoose OS" block, it is fragmented into several components as well. Some of them, like configuration, RPC, timers,

networking API, etc, will be covered further down.

Source code structure

The Mongoose OS core lives at [cesanta/mongoose-os](https://github.com/cesanta/mongoose-os) on GitHub:

- `src/` - cross-platform API
- `src/common/` - various utility functions
- `src/frozen/` - JSON parser/emitter
- `platforms/` - platform-specific code

The bulk of the functionality, however, is split into libraries. Each library is a separate GitHub repository, collected under the [mongoose-os-libs](https://github.com/mongoose-os-libs) organisation, which serves as a central repository of libraries.

When documentation is generated, all libraries are traversed and the "API Reference" part is automatically generated. The `docs:` tag in the library's `mos.yml` file specifies the documentation category and title. For example, for the ADC library located at <https://github.com/mongoose-os-libs/adc>, the `mos.yml` contains:

```
tags:  
  - docs:core:ADC  
  ...
```

That creates an [API Reference/Core/ADC](#) documentation page. The content is generate from the [README.md](#) and header files.

Boot process

The boot process is driven by a cross-platform `mgos_init.c`.

In short, the subsystems are initialised in the following order:

Native SDK init, GPIO, configuration, WiFi, platform-specific init, libraries (they can define their initialisation order), user app init function `mgos_app_init()`, and at the end - all registered `MGOS_HOOK_INIT_DONE` hooks are invoked.

The initialisation function has the following prototype:

```
enum mgos_init_result mgos_XXX_init(void);
```

It returns `MGOS_INIT_OK` on success, or any other specific numeric code on error.

If any of those init functions returns an error, the firmware reboots immediately. This is done intentionally, in order to revert back to the previous firmware in case of failed OTA update.

File system

Mongoose OS implements Virtual File System layer, VFS. That means it can attach (mount) different storage types into a single file system tree.

For example, a device can have an SPI flash storage and an SD card storage. For each storage type, a filesystem driver must be implemented. For example, it is possible to write a driver that implements a Dropbox or Google Drive storage type, and a device (e.g. ESP8266 module) can mount a Dropbox folder.

Mongoose OS provides a

[Filesystem RPC service](#)

that allows remote filesystem management - for example, you can edit files remotely.

The contents of the filesystem depends on the app and specific libraries that are used. For example, an `mjs` library which adds JavaScript support to the app, also adds a bunch of `api_*.js` files to the filesystem. Here is a typical layout:

```
conf0.json      - default app configuration, must NOT be edited manually  
conf9.json      - user-specific overrides, changed by "mos config-set" command  
index.html     - many apps define this file, which is served by a web server  
ca.pem         - added by the ca-bundle library, contains ca root certs  
init.js        - main file for the JavaScript-enabled apps  
api_*.js       - JavaScript API files  
api_*.jsc      - compiled JavaScript files
```

Main event loop

Mongoose OS contains

[Mongoose Networking Library](#) as one

of the core components. The networking library provides network protocol support, such as UDP, MQTT, etc. It constitutes the low level of Mongoose OS; it is non-blocking and event based, uses `mg_` API prefix and expects the following usage pattern:

- User creates listening connections by calling `mg_bind()` or variants
- User creates outgoing connections by calling `mg_connect()` or variants
- For all connections, listening or outgoing, user must define an event handler function
- All connections are inserted in a linked list in a `struct mg_mgr` structure, which is an event manager
- User makes an infinite event loop. On each iteration, Mongoose Networking library waits for IO on all connections. When it happens, an event handler is called for a appropriate connection

Mongoose OS does exactly that. It defines a "system" event manager instance, and runs a main event loop in a single task. That event loop dispatches events by calling event handlers. For example, `mgos_gpio_set_button_handler()` function sets up a button press event handler. When a hardware interrupt occurs, its handler queues the event, and the Mongoose OS task calls the user-defined button handler in its context.

For network connections, Mongoose OS defines wrappers for low-level `mg_` functions. These wrappers use "system" event manager and provide the reconnection functionality for the outgoing connection.

For example, low-level `mg_` API for MQTT protocol allows to create an MQTT client. If it disconnects for any reason, e.g. temporary WiFi connectivity loss, the connection closes. The `mgos_` wrapper, however, would setup a reconnection

timer with exponential backoff and re-establish the connection automatically.

This is a valuable addon to the low-level mg_ API, therefore using

mgos_ API is a good idea. Of course the low level mg_ API is also available.

You can get main event manager instance by calling `mgos_get_mgr()`

function defined in `mgos_mongoose.h`.

Note that mgos_ API, as well as mg_ API, is cross-platform. A firmware written with that API only, is portable between supported architectures, as demonstrated by many [example apps](#).

However, the native SDK API is not in any way hidden and is fully available.

For example, one could fire extra FreeRTOS tasks on platforms whose SDK use FreeRTOS. The price to pay is loss of portability.

Example footprint

Mongoose OS is highly modular - it is possible to include or exclude functionality depending on specific needs. That is implemented by the library mechanism, described later. In order to get a feeling about the resulting footprint, the table below shows measurements done on

[TI CC3220SF](#)

platform for Mongoose OS 1.18 release, built with different options.

RAM figures are measured after Mongoose OS is initialised, i.e. those numbers are what is available for the application code.

Name	Code Size	Free RAM	Notes
minimal	113k	230k	An example-no-libs-c app. Includes RTOS, TCP/UDP networking core, file system, configuration infrastructure, SNTP
minimal+mjs	160k	229k	Minimal + mJS JavaScript engine
minimal+aws	133k	230k	Minimal + AWS IoT support
minimal+gcp	159k	230k	Minimal + Google IoT Core support
js-demo	304k	225k	A default JS demo app . Includes HTTP, MQTT, WebSocket, mDNS networking, RPC subsystem, AWS IoT, Google IoT Core, JavaScript, I2C, SPI, PWM, ADC, and more - see js-demo-bundle library

RPC - Remote Procedure Calls

RPC means Remote Procedure Call. This is the way to send commands to devices and receive replies, i.e. call remote procedures.

Mongoose OS uses

[JSON-RPC 2.0](#).

Since the RPC mechanism uses JSON-RPC, that means that the "procedure", or an "RPC service" that device implements, is a function written in C/C++ or JavaScript with the following properties:

- Has a name, for example `GPIO.Toggle`,
- Takes a `params` JSON object,
- Returns a `result` JSON object.

For example, an RPC request to set GPIO pin 2 to high voltage looks like this:

```
{ "id": 1932, "method": "GPIO.Write", "params": {"pin": 2, "value": 1} }
```

The successful reply from a device looks like this:

```
{ "id": 1932, "result": true }
```

The failure reply looks like this:

```
{ "id": 1932, "error": { "code": 400, "message": "error setting pin mode" } }
```

Note that the "jsonrpc": "2.0" attribute in the request frame can be omitted.

Mongoose OS libraries implement a large set of ready-to-go RPC services, like managing hardware peripherals (GPIO, SPI, I2C), managing files, remote updates, etc. It is easy to add custom RPC services too, see [RPC Core library](#) for more details.

The JSON-RPC messages could be carried out by many different channels: serial (UART), HTTP/Restful, WebSocket, MQTT, Bluetooth. RPC API allows to add support for other channels. `mos` tool provides an easy way to call device's RPC services over the serial connection or over the network. Alternatively, RPC services can be invoked programmatically.

That means, you can write a function in C or JavaScript, and call it remotely via multiple protocols. Example: an RPC service `Sum` that adds two numbers `a` and `b`:

```
RPC.addHandler('Sum', function(args) {
  if (typeof(args) === 'object' && typeof(args.a) === 'number' && typeof(args.b) === 'number') {
    return args.a + args.b;
  } else {
    return {error: -1, message: 'Bad request. Expected: {"a":N1,"b":N2}'};
  }
});
```

See [RPC.addHandler API reference](#).

The C/C++ implementation that does the same would look something like this:

```
# include "mgos_rpc.h"
<p class="mume-header" id="include-mgos_rpch-1"></p>

static void sum_cb(struct mg_rpc_request_info *ri, void *cb_arg,
                   struct mg_rpc_frame_info *fi, struct mg_str args) {
  double a = 0, b = 0;
  if (json_scanf(args.p, args.len, ri->args_fmt, &a, &b) == 2) {
    mg_rpc_send_responsef(ri, "%.2lf", a + b);
  } else {
    mg_rpc_send_errorf(ri, -1, "Bad request. Expected: {\\"a\\":N1,\\"b\\":N2}");
  }
  (void) cb_arg;
  (void) fi;
}

// Somewhere in init function, register the handler:
mg_rpc_add_handler(mgos_rpc_get_global(), "Sum", "{a: %lf, b: %lf}", sum_cb, NULL);
```

See [mgos_rpc_add_handler API reference](#). Make sure to include the `rpc-common` library in your app's `mos.yml`.

Here is how you can call it:

Via serial connection, if a device is attached to a serial port:

```
mos --port /dev/ttyUSB0 call Sum '{"a":1, "b": 2}'  
3
```

Via WebSocket, if a device is in the local network:

```
mos --port ws://192.168.0.206/rpc call Sum '{"a":1, "b": 2}'  
3
```

Via RESTful call, if a device is in the local network:

```
curl -d '{"a":1, "b": 2}' 192.168.0.206/rpc/Sum  
3
```

Via an MQTT server, talking to an device idendified by its ID esp32_6732ac:

```
mos --port mqtt://my.mqtt.server:1883/esp32_6732ac call Sum '{"a":1, "b": 2}'  
3
```

To see a list of all RPC services implemented by a device, call `RPC.List`:

```
mos call RPC.List  
[  
    "I2C.WriteRegW",  
    "I2C.WriteRegB",  
    "I2C.ReadRegW",  
    "I2C.ReadRegB",  
    ...  
]
```

Remote management

Mongoose OS has many built-in RPC services: for managing files, accessing hardware peripherals, performing over-the-air updates, etc. In fact, the majority of `mos` command talks to a device via an RPC call.

See [Secure remote device management with Mongoose OS and AWS IoT](#) for an example of remote management over the secure AWS IoT cloud. In that case, Mongoose OS uses secure MQTT RPC channel, authenticated via the mutual TLS - per AWS IoT standards.

An ability to call RPC services via network add an incredible level of control, implementing remote management capabilities at the highest levels.

Example: scan I2C bus and return addresses of I2C peripherals from a device at IP address 192.168.1.4 using `mos` tool in a command line mode:

```
mos --port ws://192.168.1.4/rpc call I2C.Scan  
[  
    31  
]
```

Licensing

Mongoose OS is highly modular. It consists of the

core

and a large number of

libraries. The core, and most of

the libraries, are open source and released under the

[Apache 2.0 license](#) as Mongoose OS Community Edition.

Some of the libraries are closed-source, and have certain restrictions.

Those restrictions can be removed by purchasing a Mongoose OS Enterprise Edition license.

Mongoose OS Enterprise Edition Licensing process

- Make sure that `rpc-service-ota` library is used by your app
- Login to the [Mongoose License Manager](#)
- Buy any number of licenses you need. IMPORTANT: choose "mongoose-os" license type
- If your device is connected to your workstation over USB, run:

```
mos license
```

- Or, if your device is accessible remotely, run:

```
mos --port DEVICE_ADDRESS license
```

Licenses are bound to devices, and they are permanent.

The `mos license` command creates a unique license string specific for a device,

and updates `device.license` configuration parameter, which "unlocks" the device and removes restrictions. The License Manager decreases the number of available licenses and saves a unique license string. The next time `mos license` is run for the same device, an old license string is returned by the License Manager but the number of available licenses is not decreased. Thus a device, once licensed, is licensed permanently.

Libraries with restrictions

Below is the list of closed-source libraries and their restrictions.

Library	Restriction
<code>ota-common</code>	OTA only from mDash
<code>ota-http-client</code>	OTA only from mDash
<code>ota-http-server</code>	No restrictions
<code>ota-shadow</code>	OTA only from mDash
<code>cron</code>	3 cron tasks max
<code>crontab</code>	3 crontab entries max

Apps and libraries

A Mongoose OS **app** is a firmware that does something specific. It could be built and flashed on a microcontroller. For example, a

[blynk app](#) is a firmware that makes a device controllable by the [Blynk mobile app](#).

Another example is a [default app](#) that gets flashed when you press a "Flash" button on a Web UI "device control" dialog, or type `mos flash <arch>` on a terminal. That default app blinks an LED periodically, can talk to an MQTT server, and allows user to extend the logic by editing JavaScript code directly on a device filesystem.

An app can use any number of libs. A **lib** is a reusable library. It cannot be built directly into a working firmware, because it only provides an API but does not actually use that API. An app can include a lib by listing it in the `libs:` section of the [mos.yml](#) file.

`mos build` command generates code that calls library initialisation functions. Libraries are initialised in the order of their reference.

Local and remote builds

By default, a `mos build` command that builds an app's firmware, is using so-called remote build - it packs app's sources and sends them over to the Mongoose OS build machine. This is the default behavior, cause it does not require a [Docker](#) installation on the workstation.

However, if a Docker is installed, then it is possible to build locally. This is done by adding an extra `--local` flag (see below). In this case, everything is done on the local machine. This is a preferable option for the automated builds, and for those who do not want their sources leaving their workstations. Summary:

Build type	Build command
Remote (default)	<code>mos build --platform PLATFORM</code>
Local (requires Docker)	<code>mos build --platform PLATFORM --local --verbose</code>

mos.yml file format reference

`mos.yml` file drives the way Mongoose apps are built. Below is a description of the sections (keys) in this file. Libraries also have `mos.yml` files, the only difference with apps is that they have `type: lib` key and they cannot be built into a firmware. So the following applies to both apps and libraries.

author

A string, `FirstName SecondName <Email>` of the author, example:

```
author: Joe Bloggs <joe@bloggs.net>
```

build_vars

List of Makefile variables that are passed to the architecture-specific Makefile when an app is getting built. See next section for a build process

deep-dive. An example of arch-specific Makefile is:

[platforms/esp32/Makefile.build](#).

The others are in the respective directories: `fw/platforms/*/Makefile.build`.

The example below changes ESP32 SDK configuration by disabling brownout detection:

```
build_vars:  
    ESP_IDF_SDKCONFIG_OPTS: "${build_vars.ESP_IDF_SDKCONFIG_OPTS} CONFIG_BROWNOUT_DET="
```

Another example is the [dns-sd library](#) that enables DNS-SD:

```
build_vars:  
    MGOS_ENABLE_MDNS: 1
```

binary_libs

A list of `.a` libs or directories with those. Do not put trailing slashes to directory names:

```
binary_libs:  
    - mylib/mylib.a
```

cdefs

Additional preprocessor flags to pass to the compiler, example:

```
cdefs:  
    FOO: BAR
```

That gets converted into the `-DFOO=BAR` compilation option, for both C and C++ sources.

cflags, cxxflags

Modify compilation flags for C (`cflags`) and C++ (`cxxflags`). For example, by default warnings are treated as errors. This setting ignores warnings when compiling C code:

```
cflags:  
    - "-Wno-error"
```

If what you're after is defining preprocessor variables, `cdefs` makes it easier. This snippet:

```
cdefs:  
    FOO: BAR
```

Is the same as:

```
cflags:  
    - "-DFOO=BAR"  
cxxflags:  
    - "-DFOO=BAR"
```

config_schema

This can define a new configuration section for the device, and also override

a previously defined configuration entries defined elsewhere. For example, the following snippet defines a new section `foo` and overrides a default value of `mqtt.server` set by the `mqtt` library:

```
config_schema:  
- ["foo", "o", {title: "my app settings"}]  
- ["foo.enable", "b", true, {title: "Enable foo"}]  
- ["mqtt.server", "1.2.3.4:1883"]
```

description

A string, one-line short description, example:

```
description: Send BME280 temperature sensor readings via MQTT
```

filesystem

A list of files or directories with files to be copied to the device's filesystem, example:

```
filesystem:  
- fs  
- other_dir_with_files  
- foo/somepage.html
```

includes

A list of directories with C/C++ include files. Do not put trailing slash to the directory name. Example:

```
includes:  
- my_stuff/include
```

libs

Library dependencies. Each library should have an `origin` and optionally can have `name` and `version`. `origin` is a GitHub URL, like <https://github.com/mongoose-os-libs/aws> (note: it must be a repo with `mos.yml` in the repo root!).

`Name` is used to generate the code which calls library initialization function: e.g. if the lib name is `mylib`, it should have the function `bool mgos_mylib_init(void)`. Also, for local builds, `name` is used as a directory name under `deps`: that's where `mos` clones libraries.

`version` is a git tag name, or branch name, or SHA of the library's repository. If omitted, it defaults to the `libs_version` in `mos.yml`, which, in turn, defaults to the `mos` tool version. So e.g. if the `mos` tool version is 1.21, then by default it will try to use libs with the tag 1.21. Latest `mos` will use the `master` branch.

Example:

```
libs:
  # Use aws lib on the default version
  - origin: https://github.com/mongoose-os-libs/aws

  # Use aws lib on the version 1.20
  - origin: https://github.com/mongoose-os-libs/aws
    version: 1.20

  # Use the lib "mylib" located at https://github.com/bob/mylib-test1
  - origin: https://github.com/bob/mylib-test1
    name: mylib
```

name

Override app or lib name. By default, the name is set equal to the directory name.

```
name: my_cool_app
```

sources

A list of C/C++ source files or directories with those. Do not put trailing slashes to directory names:

```
sources:
  - src
  - foo/bar.c
```

tags

A list of free-form string tags, used for Web UI search.

Some tags are predefined, they place the app or library in a certain category.

Those predefined tags are: `cloud` (cloud integrations),

`hardware` (hardware peripherals or API),

`remote_management` (remote management), `core` (core functionality). Example:

```
tags:
  - cloud
  - JavaScript
  - AWS
```

Build process deep dive

When `mos build [FLAGS]` command is executed in the app directory, the following happens:

- `mos` scans `libs:` section of the `mos.yml` file and imports all libraries into the `libs` directory (`~/.mos/libs`, could be overridden by `--libs-dir ANOTHER_DIR` flag)
- Each library also has `mos.yml` file, and a library could have a `libs:` section as well - this way the library can depend on other library. `mos` imports all dependent libraries too, recursively.
- When all required libraries are imported, `mos` executes `git pull` in each of them, in order to update. That could be switched off by `--no-libs-update` flag.
- At this point, all required libraries are imported and updated.
- `mos` combines app's `mos.yml` file together with the `mos.yml` files of

all dependent libraries, merging them into one file. The order of merging

is this: if `my-app` depends on library `lib1`, and library `lib1` depends

on library `lib2`, then

`result_yml = lib2/mos.yml + lib1/mos.yml + my-app/mos.yml`. Meaning, the application's `mos.yml` has the highest priority.

- If `--local --verbose --repo PATH/TO/MONGOOSE_OS_REPO` flag is specified, then `mos` starts a local build by invoking `docker.cesanta.com/ARCH-build` docker image. That image encapsulates a native SDK for the given architecture together with Mongoose OS sources, <https://github.com/cesanta/mongoose-os>.
`mos` tool invokes `make -f fw/platforms/ARCH/Makefile.build` for the given platform. The result of this docker invocation is a `build/` directory with build artifacts and `build/fw.zip` firmware zip file which could be flashed to the device with `mos flash` command.
- If `--local` flag is not specified, packs source and filesystem files and sends them to the Mongoose OS cloud build backend at <http://mongoose.cloud>, which performs an actual build as described in the previous step, and sends back a `build/` directory with built `build/fw.zip` and artifacts.
- Generated artifacts in the `build/` directory is as follows:

```
build/fw.zip - a built firmware  
build/fs      - a filesystem directory that is put in the firmware  
build/gen     - a generated header and source files
```

How to create a new library

- The best way to develop a new library is as part of an app development.
In your app, do a local build, which creates a `deps/` directory. That is the directory where you should place your new library.
- Clone an `empty` library, which is a skeleton for the new library, into the `deps/mylib` directory (change `mylib` to whatever name you want):
`git clone https://github.com/mongoose-os-libs/empty deps/mylib`
- Create `include/mgos_mylib.h` and `src/mgos_mylib.c` files in your library:

`mgos_mylib.c:`

```
# include "mgos_mylib.h"  
<p class="mume-header" id="include-mgos_mylibh"></p>  
  
// NOTE: library init function must be called mgos_LIBNAME_init()  
bool mgos_mylib_init(void) {  
    return true;  
}
```

`mgos_mylib.h:`

```
# include "mgos.h"  
<p class="mume-header" id="include-mgosh-4"></p>
```

- You can add your library-specific API to `mgos_mylib.h` and implementation in `mgos_mylib.c`.
- In your app's `mos.yml` file, add a reference to the new library:

```
libs:  
- name: mylib
```

- Click on build button to build an app, and flash button to flash it
- Edit library source files `mylib/src`, build `myapp` until a test app works as intended.

How to port an Arduino library

- Follow the steps outlined in the previous section.
- Copy Arduino library sources into the `mylib/src` directory, and `.h` files into the `include/` directory
- Add C wrappers for the C++ API. Make it so that wrappers are FFI-able into JS : use simple types in the API, max 6 32-bit params, 2 64-bit params. See <https://github.com/cesanta/mjs#cc-interoperability>
- If you plan to add JavaScript support too, create `mjs/api_mylib.js` file with the FFI JS wrappers.
- Build / test `myapp` until it works.
- See example libraries at <https://mongoose-os.com/docs/mongoose-os/api/arduino/adafruit-ads1x15.md>

Contributing an app or library

If you would like to share your project with a community and publish it under the [Apache 2.0 license](#), please follow these steps:

- Build your app as described in the previous section, flash and test it.
- Modify `mos.yml`, set `author` field as `Your Name <your@email.address>`.
- Make sure you have a descriptive `README.md` file.
- If this is a library:
 - Create `mjs_fs/api_<name>.js` file if your library has JavaScript API.
 - If it is a port of an Arduino library, make sure you include `arduino-compat` library in `mos.yml` file, see [arduino-adafruit-ssd1306 lib](#) for an example
 - See <https://github.com/mongoose-os-libs/blynk> for the reference
 - Consider contributing an example app that uses your library
- [Start a new discussion on forum](#) with a subject `New contribution: ...`, show a link to your code on GitHub / Bitbucket / whatever, or attach a zip file with the app sources.

Commercial firmware

Below is the procedure on how to build a firmware using Mongoose OS Enterprise Edition (commercial version).

1. Acquire Mongoose OS license
2. Nominate a single GitHub account (user, not organisation) that gets an access to the private repo <https://gitub.com/cesanta/mos-libs>
3. On your build machine,
 - Clone `mos-libs` to some directory
 - Add `--local --verbose --libs-dir /path/to/mos-libs` arguments to your `mos build` command

Example command line to build a firmware:

```
mos build --platform stm32 --build-var BOARD=B-L475E-IOT01A \
--local --verbose --libs-dir ../mos-libs
```

Mongoose OS public libraries are located under the

<https://github.com/mongoose-os-libs> organisation. That org contains libraries authored by Cesanta, as well as libraries contributed by the community.

The mos-libs private repository contains only libraries authored by Cesanta. Also, the mos-libs repository contain the source code for all libraries, whereas some of the mongoose-os-libs libraries are in the binary form, built from sources with certain compilation flags that enable restrictions.

The commercial firmware must be built against the mos-libs libraries, in order to use the sources, without enabling restrictions. When you specify `--libs-dir /path/to/fetched/mos-libs` then your firmware is built correctly: Cesanta's libraries would use the locally fetched mos-libs, and all other libraries clone mongoose-os-libs.

- [Introduction](#)
- [RPC mechanism](#)
- [Device config](#)
- [Build Process](#)
- [Debugging](#)
- [OTA updates](#)
- [Security](#)
- [mos tool reference](#)
- [Licensing](#)

mbedTLS library optimisation

Mongoose OS uses mbedTLS library from ARM with several patches:

- RAM usage by each connection. By default, mbedTLS uses allocates >32k RAM for each TLS connection. Mongoose OS uses dynamic buffers, reducing RAM usage per connection down to 1k RAM.
- ATECC608A crypto chip integration. This patch makes mbedTLS to offload crypto to the hardware chip.
- Dynamic CA certificates loading. By default, mbedTLS loads all CA certificates in RAM. Our patch makes it load on demand, saving a lot of RAM. Mongoose OS stores CA certificates in ca.pem file, where you can add your server's CA certificate without blowing RAM usage.

ESP32 flash encryption

Encrypting on-flash code and data is essential:

- it protects from the reverse engineering
- it protects the device credentials, like private keys, being copied and used to get cloud access

ESP32 chip comes with built-in security features, one of which is a transparent SPI flash encryption - for details, see

[Espressif documentation](#).

Mongoose OS makes ESP32 flash encryption setup easy.

By default, Mongoose OS firmware is flashed in plain-text form:

```
mos flash esp32          # Flash Mongoose OS on ESP32
mos flash-read --arch esp32 0x190000 2000 -  # Dump filesystem area
```

The `flash-read` command dumps the flash memory into a file, and the output can show that the contents is not encrypted. Therefore, sensitive information like TLS private keys could be easily stolen from the flash.

In this case, we see a part of the device's file system, not encrypted.

In order to enable flash encryption, use `esp32-gen-key` command. It enables flash encryption for the next flashing (and sets efuses):

```
mos -X esp32-gen-key flash_encryption_key fe.key \
--esp32-enable-flash-encryption --dry-run=false
mos flash esp32 --esp32-encryption-key-file fe.key
```

That is irreversible - once flash encryption is enabled, you cannot go back.

Note the extra flag `--esp32-encryption-key-file fe.key`

for the `flash` command. From now on, a key file is required to re-flash the device.

If the key file is lost, the module can't be reflashed.

After flash encryption is enabled, the very first boot performs an encryption, which takes a while - up to a minute.

Subsequent boots will be normal, not doing any encryption.

Once the flash is encrypted, one can verify it using `flash-read` command to ensure there no plain-text parts are present:

```
mos flash-read --arch esp32 0x190000 2000 -
```

It is recommended to use a separate key for each device. The efuses must be set to enable encryption (this is done automatically by `esp32-gen-key`) and can be verified by running:

```
mos esp32-efuse-get
```

A device with encryption enabled should show `flash_crypt_cnt : 0x01`

ATECC608A crypto chip

Often, IoT boards provide no built-in flash protection mechanism.

Anyone with a physical access to the device can read the whole flash, including any sensitive information like TLS private keys.

Crypto chips are designed to mitigate that.

Their main function is provide storage for private keys, which cannot be read.

Private keys are stored inside the crypto chip, and all the crypto operations that require private key, are offloaded to the crypto chip which performs the operation and gives the result back.

ATECC608A crypto chip

is designed with additional hardware protection mechanisms to make key extraction difficult. It is an impressive piece of hardware with many layers of protection, and important enough it is quite inexpensive, costing less than 80 cent a piece.

Wiring (ESP8266 NodeMCU example)

Get ATECC608A - either as an ATCRYPTOAUTH-XPRO board which requires no soldering, or a bare-bones ATECC608A which requires soldering.

Function	ATECC608A pin	ESP8266 pin	NodeMCU pin	ATCRYPTOAUTH pin
SDA	5	10 (GPIO12)	D6	11 (yellow)
SCL	6	9 (GPIO14)	D5	12 (white)

GND	4	Any GND	Any GND	19 (black)
VCC	8	Any 3V3	Any 3V3	20 (red)

Wiring for ATCRYPTOAUTH-XPRO:

Wiring for the bare-bones ATECC608A:

Setup guide

Mongoose OS has native support for

[ATECC608A](#) security chip.

This section is a quick guide to get it up and running.

For a more detailed reference, especially of chip configuration, please refer to Microchip documentation.

1. The chips leave the factory unconfigured. Blank chip will be detected but crypto operations failing with code 0xf4. Microchip provides their own configuration tools but `mos` includes basic commands to get and set configuration as a YAML file. For development you can use our [sample configuration](#).

To set it, use the following `mos` commands:

```
mos atca-set-config atca-test-config.yaml --dry-run=false
mos atca-lock-zone config --dry-run=false
mos atca-lock-zone data --dry-run=false
```

Note: This only needs to be done once and once locked, chip configuration cannot be changed anymore. You can dump chip's configuration with `mos atca-get-config --format=yaml`.

Note 2: Sample config is very permissive and is only suitable for development and testing, NOT for production deployments. Please refer to Microchip's manual and other documentation to come up with more secure configuration (we may be able to assist with that too - ask a question on [our forum](#)).

2. Generate a cert and key as normal. An example below shows a self-signed certificate, but of course it doesn't have to be. The important thing is that it's a ECDSA certificate using P256 curve, since that is what the chip supports.

```
openssl ecparam -out ecc.key.pem -name prime256v1 -genkey
openssl req -new -subj \
  "/C=IE/L=Dublin/O=ACME Ltd/OU=Testing/CN=test.acme.com" \
  -sha256 -key ecc.key.pem -text -out ecc.csr tmpl
openssl x509 -in ecc.csr.pem -text -out ecc.crt.pem \
  -req -signkey ecc.key.pem -days 3650
```

3. Write the generated key into the device. Assuming you are using our sample configuration described in the previous section, this is a two-step process:

3.1. Generate and set the key encryption key in slot 4

```
openssl rand -hex 32 > slot4.key
mos -X atca-set-key 4 slot4.key --dry-run=false
AECC508A rev 0x5000 S/N 0x012352aad1bbf378ee, config is locked, data is locked
Slot 4 is a non-ECC private key slot
SetKey successful.
```

3.2. Set the actual ECC key in slot 0

```
mos -X atca-set-key 0 ecc.key.pem --write-key=slot4.key --dry-run=false
AECC508A rev 0x5000 S/N 0x012352aad1bbf378ee, config is locked, data is locked

Slot 0 is a ECC private key slot
Parsed EC PRIVATE KEY
Data zone is locked, will perform encrypted write using slot 4 using slot4.key
SetKey successful.
```

4. Upload the certificate to the device

```
mos put ecc.crt.pem
```

5. Set HTTP server configuration to use the uploaded certificate and private key from device's slot 0:

```
mos config-set http.listen_addr=:443 \
  http.ssl_cert=ecc.crt.pem http.ssl_key=ATCA:0
Getting configuration...
Setting new configuration...
Saving and rebooting...
```

At startup you should see in the device's log:

```
mgos_sys_config_init_http HTTP server started on [443] (SSL)
```

And when connecting with the browser:

```
ATCA:2 ECDH get pubkey ok
ATCA:0 ECDSA sign ok
ATCA:2 ECDH ok
```

Configuring TLS on device's HTTP/WS server

The system HTTP server has the following configuration options (we've added comments to the `mos` tool output):

```
mos config-get http
{
  "enable": true,          # Set to false to disable default HTTP server
  "hidden_files": "",      # Glob pattern for files to hide from serving
  "listen_addr": "80",      # Port to listen on
  "ssl_ca_cert": "",       # CA certificate for mutual TLS authentication
  "ssl_cert": "",          # Certificate file
  "ssl_key": "",           # Private key file
  "upload_acl": "*"        # ACL for which files can be uploaded via /upload
}
```

In order to setup one-way SSL/TLS on the system HTTP server, create a certificate, upload the certificate and the key file to the device, and change HTTP server configuration.

How to create a self-signed certificate

```
openssl req -nodes -new -x509 -keyout key.pem -out cert.pem
mos put cert.pem
mos put key.pem
mos config-set http.listen_addr=443 http.ssl_key=key.pem http.ssl_cert=cert.pem
mos wifi WIFI_NET WIFI_PASS
curl -k https://IP_ADDRESS      # Test it !
```

If you want to use mutual (two-way) TLS with the device, follow this procedure to use a self-signed certificate:

Self-signed certificate for mutual TLS

```
# Common parameters
<p class="mume-header" id="common-parameters"></p>

SUBJ="/C=IE/ST=Dublin/L=Docks/O=MyCompany/CN=howdy"

# Generate CA
<p class="mume-header" id="generate-ca"></p>

openssl genrsa -out ca.key 2048
openssl req -new -x509 -days 365 -key ca.key -out ca.crt \
-subj /C=IE/ST=Dublin/L=Docks/O=mos/CN=me

# Generate client cert
<p class="mume-header" id="generate-client-cert"></p>

openssl genrsa -out client.key 2048
openssl req -new -key client.key -out client.csr -subj $SUBJ
openssl x509 -req -days 365 -in client.csr -CA ca.crt \
-CAkey ca.key -set_serial 01 -out client.crt

# Generate server cert
<p class="mume-header" id="generate-server-cert"></p>

openssl genrsa -out server.key 2048
openssl req -new -key server.key -out server.csr -subj $SUBJ
openssl x509 -req -days 365 -in server.csr -CA ca.crt \
-CAkey ca.key -set_serial 01 -out server.crt
```

When done, copy generated files to the device and reconfigure the server:

```
# Upload server key, cert & ca cert to the device
<p class="mume-header" id="upload-server-key-cert-ca-cert-to-the-device"></p>

mos put ca.crt
mos put server.key
mos put server.crt

# Update HTTP server settings to use mutual TLS
<p class="mume-header" id="update-http-server-settings-to-use-mutual-tls"></p>

mos config-set http.ssl_ca_cert=ca.crt http.ssl_cert=server.crt \
http.ssl_key=server.key http.listen_addr=443
```

From that point on, the device should be accessible via secure Websocket:

```
# Configure WiFi
<p class="mume-header" id="configure-wifi"></p>

mos wifi WIFI_NET WIFI_PASSWORD
mos --cert-file client.crt \
--key-file client.key \
--port wss://IPADDR/rpc \
call RPC.List
```

Device as a network client

A connected device can be a network client, or network server, or both. For example, if a device provides a RESTful interface, it acts as a network server. If a device connects to an MQTT server, it acts as a network client.

Avoid running a network server on your device for the following reasons:

- It is easy to DoS the device by creating many network connections. Say, a device has 40k of free RAM, and each connection takes 10k, then 4 connections is enough for the denial of service
- A device must implement authentication and authorisation mechanisms that are potentially vulnerable
- A network service code may be vulnerable
- If TLS is used for communication, the connection setup time could be large because of the slow CPU, leading to delays and bad user experience
- TLS certificate management for the local communication could be non-trivial

On the other hand, when a device acts as a client, these problems disappear:

- It is impossible to hack into the device directly because it does not expose any network endpoint
- A device does not care about authentication and authorisation - it is all handled on the cloud side, and secure services like Google IoT Core or AWS IoT would be a good choice for the cloud backend
- The only entity a device should trust is a cloud backend, which is handled by the industry-standard TLS
- No need to keep many network connections, cause a single secure connection to the cloud backend is enough for both management and data flows. This saves precious resources

Securing RPC

RPC is a Mongoose OS mechanism for remote management. Every time `mos` tool is fired in a command line mode or in the GUI mode, RPC is used to query the device - over serial or remotely. Displaying list of files, showing what hardware platform it is, amount of free RAM, toggling GPIO - it is all RPC.

When a new device is flashed, it is in a default state when all RPCs are allowed for everybody. Several mechanisms are available to restrict the access to RPC services:

- Enable authentication - which users are allowed
- Enable authorisation - what allowed users can do
- Disable RPC for the given channel
- Disable all RPC functionality

Enable authentication

Mongoose OS implements authentication using Digest mechanism, the same as [HTTP Digest authentication](#)

employs. It works like this - true for both HTTP Digest and Mongoose OS RPC:

- Client sends a request
- If authentication is not enabled, server replies
- If authentication is enabled and client request has authentication data, server replies
- If authentication is enabled and client request does not have authentication data, server sends back an error requesting authentication with a random nonce to prevent replay attacks
- Client repeats the request with the authentication data created with the

server's nonce

In the case of HTTP, an authentication data is sent in the `Authorization:`

HTTP header. In the case of RPC, an authentication data is sent as an additional `auth` key in the RPC frame.

The authentication is enabled by creating a passwords file in the `htdigest` standard format and setting it in the configuration.

The format of the passwords file is the same as for the HTTP Digest authentication, and it could be managed by the Apache's `htdigest`.

This is an example that creates a passwords file with user `joe`, uploads that file to the device, and configures RPC to use it:

```
htdigest -c rpc_auth.txt myproduct joe
mos put rpc_auth.txt
mos config-set rpc.auth_domain=myproduct
mos config-set rpc.auth_file=rpc_auth.txt
```

Enable authorisation

RPCs that are authenticated, could be checked against the ACL file to determine whether a given authenticated user can access an RPC:

```
mos put rpc_acl.json
mos config-set rpc.acl_file=rpc_acl.json
```

The format of the ACL is similar to the config management ACL. Here is an example of `rpc_acl.json` file:

```
[
  {"method": "FS.*", "acl": "+user1,-user2"},
  {"method": "*", "acl": "-*"}
]
```

Disable RPC for given transport

If the transport is disabled, that's it!

RPC transports are serial, HTTP/RESTful,

WebSocket, MQTT, Bluetooth. This disables HTTP transport:

```
mos config-set rpc.http.enable=false
```

Disable all RPC functionality

This is the most radical method. Remove `rpc-common` library, and all other `rpc-*` libraries from your `mos.yml`. That totally removes RPC functionality from the firmware.

Note, however, that the RPC security is determined by the security of its transport. For example, RPC over AWS IoT service uses secure, authenticated AWS IoT mechanism, utilising mutual TLS and elaborate policy infrastructure. Thus, encryption, authentication and authorisation of the RPC channel is provided by AWS. Such an RPC channel is highly secure. The opposite example would be an RPC over plain HTTP/RESTful, not authenticated and not locked by authorisation, wide open to the world.

An alternative to RPC for the remote management could be a mechanism

provided by the cloud backend - for example, AWS IoT device shadow, or Azure device twin, or Google IoT Core config/state objects, etc.

Device Management Dashboard

Please see [Dashboard Introduction](#).

OTA - Over The Air firmware update

OTA using `mos ota` command

This is the most universal OTA method. It works over any transport: serial connection, BLE, MQTT, Websocket, etc.

```
mos --port DEVICE_ADDRESS ota FIRMWARE_FILENAME_OR_URL
```

OTA using management dashboard file upload

This method is the simplest one. Build, or download a new firmware. Then, open a management dashboard, select devices you want to update, and upload a .zip file using the Web UI:

The in-progress is visible on the dashboard, regardless of the OTA method:

OTA by calling `OTA.Update` device RPC

```
mos --port DEVICE_ADDRESS ota FIRMWARE_FILENAME_OR_URL
```

OTA using `OTA.Update` RPC call

This method works by downloading a new firmware over HTTP.

NOTE: this method requires professional subscription.

```
mos --port DEVICE_ADDRESS call OTA.Update '{"url": "URL_TO_ZIP"}'
```

OTA using device shadow

This method works by setting a `desired.ota.url` shadow parameter.

As soon as a device comes online, it picks up the change, and downloads the new firmware. NOTE: this method requires professional subscription.

This method works on [mDash](#), AWS IoT, Microsoft Azure. Just change the device shadow manually or programmatically - here is an example of the mDash online shadow editor:

OTA using HTTP POST

This method is for development. It works only if the device is directly visible. In order to enable HTTP POST OTA handler, include

[ota-http-server](#)

library in your `mos.yml`. Then, you

can build a new firmware and push it using this command:

```
curl -v -F file=@build/fw.zip http://IP_ADDR/update
```

OTA architecture overview

OTA implementation in Mongoose OS has 3 layers:

Layer 1. The lowest layer, implemented by the [ota-common](#)

library. The API functions that do the work are

`updater_context_create()` to start an update, a sequence of

`updater_process()` to apply the

next chunk of firmware, and `updater_finish()` to finish the update.

Layer 2. The RPC service that provides a remote management API for the

low-level OTA API. It is implemented by the [rpc-service-ota](#)

library. The RPC API are of two kinds: push and pull.

- `OTA.Update` is a pull mechanism. Works only via HTTP.
You pass a URL to the new .zip file as a parameter of the RPC call,
the .zip file gets downloaded and low-level API calls apply it.
- `OTA.{Begin,Write,End}` is a push mechanism. It works over any transport,
for example, BLE, or MQTT. They are calling corresponding
low-level API.

Layer 3. Helper libraries and tools that invoke the 2nd RPC layer,

- `mos ota` command calls the "push" RPC, and can work over any transport by specifying the `--port ...` parameter. That is the most universal method.
- [ota-http-client](#)
library is able to fetch firmware from a URL. This library is in fact used by the `OTA.Update` RPC implementation. Also, this library is able to periodically poll a given URL for the new version of firmware, and auto-update.
- [ota-http-server](#) library
registers and `/update` URI handler directly on a device. You can push the new firmware over the HTTP POST. This method is for the development purposes: `curl -v -i -F filedata=@fw.zip http://IPADDR/update`
- [ota-shadow](#) library
observes `desired.ota.url` shadow changes. If it gets a new URL, it triggers the `OTA.Update` with that URL. This method works for the offline devices - you can change the shadow of the number of devices, and they will get updated as soon as they come online.

Commit timeout

When a new firmware boots, it can hang or crash for various reasons.

It can happen on one of a thousands devices, but still you don't want

it to happen on a customer site. The concept of commit timeout is

a solution for this problem that makes OTA process reliable - you can

push a very untested, dodgy firmware on a production device, and it will

roll back if there is any problem.

It works this way: first of, the OTA process never mutates the existing flash partition. OTA-ed firmware and filesystem gets copied to the new partition without touching an existing one.

Secondly, a new firmware boots from a flash partition that is marked "dirty". When a boot loaded sees a "dirty" flag, it starts a hardware timer that triggers after `commit_timeout` seconds. If a "dirty" flag is still present when a timer kicks in, a boot loader regards a firmware bad and reboots back into an old partition.

Thus, a commit timeout feature gives a "verification period" during which a decision is made whether a new firmware is good or not.

Clearing the dirty flag is done by the `OTA.Commit` call, and could be done either manually or automatically. If the firmware hangs, or crashes before the dirty flag is cleared, the next reboot will roll back.

NOTE: the commit timeout support requires professional subscription.

Implementation details

The main design principle of the OTA mechanisms is reliability: never end up with a bricked device, roll back on any failure. Therefore, an OTA process never updates firmware code or data in-place. Any failure (e.g. power loss) can end up in broken device. Thus, an OTA mechanism uses independent self-contained flash partitions to hold firmware images (code and data), and an intelligent boot loader makes a decision which partition to boot:

Here is a high level overview of the OTA procedure:

1. OTA is triggered via one of the many supported methods:
HTTP POST request,
[periodic timer that polls well known location](#),
[AWS IoT device shadow change](#),
an `OTA.Update` [RPC command](#), or other.
You can create your method using an [OTA API](#).
2. A separate flash partition is created to hold a new firmware image - code and data (root filesystem).
3. A new firmware image is downloaded to the new flash partition. Any failure during that process aborts an OTA.
4. When new firmware image is successfully copied,
 - All files from the old FS that do not exist in the new FS, are moved to the new FS. This is an important mechanism of preserving user data and device-specific configuration, like `conf2.json` - `conf9.json` configuration files, or any other files. Remember: if a firmware image contains a file, it'll override an existing file during OTA.
Never put files like `conf9.json` in your firmware.
 - Boot loader configuration is updated, saying that a new partition exists and the boot loader must boot from it. A new partition is marked dirty, and the "commit interval" time is stored in the boot configuration.
5. Device reboots. Boot loader boots the new partition. It figures out from the boot configuration that that partition is dirty, unsafe, because the "commit" flag is not set. Therefore it starts the hardware timer that will

fire after the "commit interval", and executes the new image.

6. The new image starts, performs a usual boot sequence. At some point `mgos_upd_commit()` is called, which sets a "commit" flag in the boot config, marking this firmware "OK". A commit call could be done automatically after the health-checks, or manually by the user.
If the commit is not made, a boot config still has "commit" flag not set.
7. A boot loader timer handler kicks in. It checks the commit flag. If it is set, it silently exits. If not set, i.e. the firmware is still dirty, the rollback is performed: the image to boot, and commit flag are set to their previous values, and device reboots.

The in-depth example of the OTA on CC3200 is given at

[embedded.com article - Updating firmware reliably](#)

Boot configuration section

If the boot config is stored in only one location, it makes it susceptible to failure during updates, which are usually performed as a read-erase-write operation: a reboot after erase and before write is complete could render device unbootable. The time between the two is short, but we set out to make our update process safe at all points, so we have to deal with it. The way we do it by using two config files with versioning, or sequencing. A sequencer is a monotonically decreasing number, so of the two files the one with smaller sequencer is more recent - on figure 2, config 1 is selected as active because it has smaller sequencer.

When writing a new config file, we always use the currently inactive (older) slot and it will not become newer until it is written - erased config will be older than any valid one because erased NOR flash is filled with all 1s:

mos tool reference

Using --port option

`mos` tool connects to the device specified by `--port` flag, which is set to `auto` by default. That means, `mos` auto-detects the serial port for the device. You can specify this value manually. It could be a serial device, e.g. `--port COM3` on Windows or `--port /dev/ttyUSB0` on Linux.

It is possible to set `--port` value to be a network endpoint instead of serial port. Device listens for commands on serial, Websocket, and MQTT transports (unless they are disabled). Therefore, `--port ws://IP_ADDR/rpc` connects to the remote device via Websocket, and `--port mqtt://MQTT_SERVER/DEVICE_ID/rpc` via the MQTT protocol. That gives an ability to use `mos` tool as a remote device management tool.

Using environment variables to set default option values

The default values for any `mos` flag could be overridden via the environment variable `MOS_FLAGNAME`. For example, to set the default value for `--port` flag, export `MOS_PORT` variable - on Mac/Linux, put that into your `~/.profile`:

```
export MOS_PORT=YOUR_SERIAL_PORT # E.g. /dev/ttyUSB0
```

Boards wiring

In some cases, for example if you're using a bare-bones ESP8266 module instead of a development board, you need to perform extra steps to switch the module between flashing and firmware boot state. This table provides a summary:

Platform	Wiring Notes
bare bones ESP8266	flash via UART: GPIO15 LOW, GPIO0 LOW, GPIO2 HIGH boot from flash: GPIO15 LOW, GPIO0 HIGH, GPIO2 HIGH boot from SD: GPIO15 HIGH
bare bones ESP32	flash via UART: GPIO0 LOW boot from flash: GPIO0 HIGH
CC3200 launchpad	connect J8 to SOP2 (see guide)

Versioning

The `mos` tool could be self-updated via the Web UI or via the console command `mos update`. The `mos` tool version also influences the firmware build: the libraries that are used during the build correspond to the `mos` version. There are 3 ways you can stay updated:

- Pin to a specific version, e.g. `mos update 1.18`. This is the most stable approach, as nothing gets changed in this case
 - Pin to the "release" channel, `mos update release`. This is the default. Released are created once in 1-2 weeks
 - Pin to the "latest" channel, `mos update latest`. Get the most latest updates, but experience breakages sometimes
- [IL0373 SPIIL0398 SPIIL91874 SPISSD1608 SPISSD1675 SPI](#)
[IL0373 SPIIL0398 SPIIL91874 SPISSD1608 SPISSD1675 SPI](#)

IL0373 SPIIL0398 SPIIL91874 SPISSD1608 SPISSD1675 SPI

Github Repo	C Header	C source	JS source
mongoose-os-libs/arduino-adafruit-epd	Adafruit_EPD.h		api_arduino_adafruit_epd.js

Introduction

This is a port of or rather a wrapper around the [Adafruit EPD Library](#) for [Mongoose OS](#).

Please check [Adafruit EPD Library](#) for additional information and also please note the original [README](#).

Notes:

- this was moved from [bbilger/mgos-arduino-adafruit-epd](#) which was forked from [adafruit/Adafruit_EPD](#)
- EPD stands for e-paper display.

Usage

Examples

There are three examples that show how to use this library:

- [C usage example](#)
- [JavaScript or rather mJS usage example](#)
- [C++ usage example](#)

C

```
# include "mgos_aepd.h"
<p class="mume-header" id="include-mgos_aepdh"></p>

...
// get the e-paper display as configured in mos.yml
Adafruit_EPD *epd = mgos_aepd_get_global();
// methods of the Adafruit_EPD C++ class
// can be invoked through "mgos_aepd_<snake_case_method_name>(epd, ...)"
mgos_aepd_some_method(epd, ...);
...
```

JavaScript / mJS

```
load('api_arduino_adafruit_epd.js');
// get the e-paper display as configured in mos.yml
let epd = ArduinoAdafruitEpd.createGlobal();
// the JavaScript object has the same / most methods the original Adafruit_EPD C++ class has
epd.someMethod(...);
```

C++

```
# include "Adafruit_EPD.h"
<p class="mume-header" id="include-adafruit_epdh"></p>

# include "mgos_aepd.h"
<p class="mume-header" id="include-mgos_aepdh-1"></p>

...
// get the e-paper display as configured in mos.yml
Adafruit_EPD *epd = mgos_aepd_get_global();
// nothing fancy here, since one can use the original Adafruit_EPD C++ class directly
epd.someMethod(...);
```

Configuration

It's optional to setup the display using the following config but unless you are using more than one display this is probably what you want.

config	type	default	required	comment
aepd.enable	bool	false	-	If enabled, then the display will be setup automatically by the library and the rest of the config must be valid.
aepd.begin	bool	true	-	Calls begin on the display automatically on start.

aepd.driver	string	empty	true	The driver to use for the connected display. At the moment the following drivers are implemented: "IL0373", "IL0398", "IL91874", "SSD1608", "SSD1675".
aepd.width	int	-1	true	Display width in pixels.
aepd.height	int	-1	true	Display height in pixels.
aepd.epd_spi_cs_index	int	-1	true	spi.csX_gpio index for the EDP, 0, 1 or 2.
aepd.sram_spi_cs_index	int	-1	false	spi.csX_gpio index for the SRAM, 0, 1 or 2, or -1 to not use SRAM.
aepd.epd_dc_gpio	int	-1	true	EPD DC GPIO.
aepd.epd_reset_gpio	int	-1	false	EPD reset GPIO or -1
aepd.debug	bool	false	-	At the moment it only prevents a bootloop on misconfiguration.

If you whish to not setup the display via configuration, then you can set it up in C via `mgos_aepd_create`, in C++ by directly instantiating a concrete subclass of `Adafruit_EPD` (e.g. `Adafruit_SSD1675`), and in JS you are on your own but you can for example "ffi" `mgos_aepd_create`.

License

This library is - like the original one - licensed under [MIT license](#).

writeRAMCommand

```
virtual uint8_t writeRAMCommand(uint8_t index) = 0;
```

setRAMAddress

```
virtual void setRAMAddress(uint16_t x, uint16_t y) = 0;
```

powerUp

```
virtual void powerUp(void) = 0;
```

update

```
virtual void update(void) = 0;
```

powerDown

```
virtual void powerDown(void) = 0;
void hardwareReset(void);
```

JS API

- Core
- Cloud services
- Networking
- Drivers
- Arduino compat
- RPC
- Misc

JSON

Github Repo	C Header	C source	JS source
cesanta/frozen	frozen.h	frozen.c	

(*json_walk_callback_t)

```
typedef void (*json_walk_callback_t)(void *callback_data, const char *name,
                                    size_t name_len, const char *path,
                                    const struct json_token *token);
```

Callback-based SAX-like API.

Property name and length is given only if it's available: i.e. if current event is an object's property. In other cases, `name` is `NULL`. For example, name is never given:

- For the first value in the JSON string;
- For events `JSON_TYPE_OBJECT_END` and `JSON_TYPE_ARRAY_END`

E.g. for the input `{ "foo": 123, "bar": [1, 2, { "baz": true }] }`, the sequence of callback invocations will be as follows:

- type: `JSON_TYPE_OBJECT_START`, name: `NULL`, path: "", value: `NULL`
- type: `JSON_TYPE_NUMBER`, name: `"foo"`, path: `".foo"`, value: `"123"`
- type: `JSON_TYPE_ARRAY_START`, name: `"bar"`, path: `".bar"`, value: `NULL`
- type: `JSON_TYPE_NUMBER`, name: `"0"`, path: `".bar[0]"`, value: `"1"`
- type: `JSON_TYPE_NUMBER`, name: `"1"`, path: `".bar[1]"`, value: `"2"`
- type: `JSON_TYPE_OBJECT_START`, name: `"2"`, path: `".bar[2]"`, value: `NULL`
- type: `JSON_TYPE_TRUE`, name: `"baz"`, path: `".bar[2].baz"`, value: `"true"`
- type: `JSON_TYPE_OBJECT_END`, name: `NULL`, path: `".bar[2]"`, value: `{"baz": true}`
- type: `JSON_TYPE_ARRAY_END`, name: `NULL`, path: `".bar"`, value: `[1, 2, { "baz": true }]`
- type: `JSON_TYPE_OBJECT_END`, name: `NULL`, path: `""`, value: `{"foo": 123, "bar": [1, 2, { "baz": true }] }`

json_walk

```
int json_walk(const char *json_string, int json_string_length,
              json_walk_callback_t callback, void *callback_data);
```

Parse `json_string`, invoking `callback` in a way similar to SAX parsers; see `json_walk_callback_t`.

Return number of processed bytes, or a negative error code.

json_printf

```
int json_printf(struct json_out *, const char *fmt, ...);
int json_vprintf(struct json_out *, const char *fmt, va_list ap);
```

Generate formatted output into a given sting buffer.

This is a superset of printf() function, with extra format specifiers:

- %B print json boolean, true or false. Accepts an int.
- %Q print quoted escaped string or null. Accepts a const char *.
- %.*Q same as %Q, but with length. Accepts int, const char *
- %V print quoted base64-encoded string. Accepts a const char *, int.
- %H print quoted hex-encoded string. Accepts a int, const char *.
- %M invokes a json_printf_callback_t function. That callback function can consume more parameters.

Return number of bytes printed. If the return value is bigger than the supplied buffer, that is an indicator of overflow. In the overflow case, overflowed bytes are not printed.

json_fprintf

```
int json_fprintf(const char *file_name, const char *fmt, ...);
int json_vfprintf(const char *file_name, const char *fmt, va_list ap);
```

Same as json_printf, but prints to a file.

File is created if does not exist. File is truncated if already exists.

json_asprintf

```
char *json_asprintf(const char *fmt, ...);
char *json_vasprintf(const char *fmt, va_list ap);
```

Print JSON into an allocated 0-terminated string.

Return allocated string, or NULL on error.

Example:

```
char *str = json_asprintf("{a:%H}", 3, "abc");
printf("%s\n", str); // Prints "616263"
free(str);
```

json_printf_array

```
int json_printf_array(struct json_out *, va_list *ap);
```

Helper %M callback that prints contiguous C arrays.

Consumes void *array_ptr, size_t array_size, size_t elem_size, char *fmt

Return number of bytes printed.

json_scanf

```
int json_scanf(const char *str, int str_len, const char *fmt, ...);
int json_vscanf(const char *str, int str_len, const char *fmt, va_list ap);
```

Scan JSON string str, performing scanf-like conversions according to fmt.

This is a scanf() - like function, with following differences:

1. Object keys in the format string may be not quoted, e.g. "{key: %d}"
2. Order of keys in an object is irrelevant.
3. Several extra format specifiers are supported:

- %B: consumes `int *` (or `char *`, if `sizeof(bool) == sizeof(char)`), expects boolean `true` or `false`.
- %Q: consumes `char **`, expects quoted, JSON-encoded string. Scanned string is malloc-ed, caller must free() the string.
- %V: consumes `char **`, `int *`. Expects base64-encoded string. Result string is base64-decoded, malloced and NUL-terminated. The length of result string is stored in `int *` placeholder. Caller must free() the result.
- %H: consumes `int *, char **`. Expects a hex-encoded string, e.g. "fa014f". Result string is hex-decoded, malloced and NUL-terminated. The length of the result string is stored in `int *` placeholder. Caller must free() the result.
- %M: consumes custom scanning function pointer and `void *user_data` parameter - see `json_scanner_t` definition.
- %T: consumes `struct json_token *`, fills it out with matched token.

Return number of elements successfully scanned & converted.

Negative number means scan error.

(*json_scanner_t)

```
typedef void (*json_scanner_t)(const char *str, int len, void *user_data);
```

json_scanf's %M handler

json_scanf_array_elem

```
int json_scanf_array_elem(const char *s, int len, const char *path, int index,
                         struct json_token *token);
```

Helper function to scan array item with given path and index.

Fills `token` with the matched JSON token.

Return -1 if no array element found, otherwise non-negative token length.

json_unescape

```
int json_unescape(const char *src, int slen, char *dst, int dlen);
```

Unescape JSON-encoded string src,slen into dst, dlen.

src and dst may overlap.

If destination buffer is too small (or zero-length), result string is not written but the length is counted nevertheless (similar to snprintf).

Return the length of unescaped string in bytes.

json_escape

```
int json_escape(struct json_out *out, const char *str, size_t str_len);
```

Escape a string `str`, `str_len` into the printer `out`.

Return the number of bytes printed.

json_fread

```
char *json_fread(const char *file_name);
```

Read the whole file in memory.

Return malloc-ed file content, or NULL on error. The caller must free().

json_setf

```
int json_setf(const char *s, int len, struct json_out *out,
              const char *json_path, const char *json_fmt, ...);
```

Update given JSON string `s`,`len` by changing the value at given `json_path`.

The result is saved to `out`. If `json_fmt` == NULL, that deletes the key.

If path is not present, missing keys are added. Array path without an index pushes a value to the end of an array.

Return 1 if the string was changed, 0 otherwise.

Example: `s` is a JSON string { "a": 1, "b": [2] }

```
json_setf(s, len, out, ".a", "7"); // { "a": 7, "b": [ 2 ] }
json_setf(s, len, out, ".b", "7"); // { "a": 1, "b": 7 }
json_setf(s, len, out, ".b[]", "7"); // { "a": 1, "b": [ 2, 7 ] }
json_setf(s, len, out, ".b", NULL); // { "a": 1 }
```

json_prettyify

```
int json_prettyify(const char *s, int len, struct json_out *out);
```

Pretty-print JSON string `s`,`len` into `out`.

Return number of processed bytes in `s`.

json_prettyify_file

```
int json_prettyify_file(const char *file_name);
```

Prettify JSON file `file_name`.

Return number of processed bytes, or negative number of error.

On error, file content is not modified.

json_next_key

```
void *json_next_key(const char *s, int len, void *handle, const char *path,
                     struct json_token *key, struct json_token *val);
```

Iterate over an object at given JSON path.

On each iteration, fill the `key` and `val` tokens. It is OK to pass NULL

for `key`, or `val`, in which case they won't be populated.

Return an opaque value suitable for the next iteration, or NULL when done.

Example:

```
void *h = NULL;
struct json_token key, val;
while ((h = json_next_key(s, len, h, ".foo", &key, &val)) != NULL) {
    printf("[%.*s] -> [%.*s]\n", key.len, key.ptr, val.len, val.ptr);
}
```

json_next_elem

```
void *json_next_elem(const char *s, int len, void *handle, const char *path,
                     int *idx, struct json_token *val);
```

Iterate over an array at given JSON path.

Similar to `json_next_key`, but fills array index `idx` instead of `key`.

Time

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_time.h	mgos_time.c	

mgos_uptime

```
double mgos_uptime(void);
```

Get number of seconds since last reboot

mgos_uptime_micros

```
int64_t mgos_uptime_micros(void);
```

Get number of microseconds since last reboot

mgos_time_micros

```
int64_t mgos_time_micros(void);
```

Get wall time in microseconds.

mgos_strerror

```
int mgos_strerror(char *s, int size, char *fmt, int time);
```

Format `time` according to a `strftime()`-conformant format.

Write the result into the `s,size` buffer. Return resulting string length.

mgos_settimeofday

```
int mgos_settimeofday(double time, struct timezone *tz);
```

Like standard `settimeofday()`, but uses `double` seconds value instead of

`struct timeval *tv`. If time was changed successfully, emits an event

`MGOS_EVENT_TIME_CHANGED`.

Net events

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_net.h	mgos_net.c	api_net.js

Low-level network configuration API.

Contains definitions of the configuration state. Allows to set up an event handler that tracks state changes: when the network connectivity is lost, established, etc.

MGOS_EVENT_BASE

```
# define MGOS_EVENT_GRP_NET MGOS_EVENT_BASE('N', 'E', 'T')
<p class="mume-header " id="define-mgos_event_grp_net-mgos_event_basen-e-t"></p>
```

Event group which should be given to `mgos_event_add_group_handler()` in order to subscribe to network events.

Example:

```
static void my_net_ev_handler(int ev, void *evd, void *arg) {
    if (ev == MGOS_NET_EV_IP_ACQUIRED) {
        LOG(LL_INFO, ("Just got IP!"));
        // Fetch something very useful from somewhere
    }
    (void) evd;
    (void) arg;
}

// Somewhere else:
mgos_event_add_group_handler(MGOS_EVENT_GRP_NET, my_net_ev_handler, NULL);
```

mgos_net_get_ip_info

```
bool mgos_net_get_ip_info(enum mgos_net_if_type if_type, int if_instance,
                         struct mgos_net_ip_info *ip_info);
```

Retrieve IP configuration of the provided interface type and instance number, and fill provided `ip_info` with it. Returns `true` in case of success, false otherwise.

mgos_net_ip_to_str

```
char *mgos_net_ip_to_str(const struct sockaddr_in *sin, char *out);
```

Converts address to dotted-quad NUL-terminated string.

`out` must be at least 16 bytes long.

Returns the `out` pointer.

mgos_net_str_to_ip

```
bool mgos_net_str_to_ip(const char *ips, struct sockaddr_in *sin);
```

Parses dotted-quad NUL-terminated string into an IPv4 address.

mgos_net_str_to_ip_n

```
bool mgos_net_str_to_ip_n(const struct mg_str ips, struct sockaddr_in *sin);
```

Parses dotted-quad NUL-terminated string into an IPv4 address.

mgos_get_nameserver

```
char *mgos_get_nameserver(void);
```

Returns nameserver address. The caller should `free()` it. Returns NULL in case of no DNS.

JS API

Net.ctos

```
Net.ctos(conn, local, ip, port)
```

Convert address of a connection `conn` to string. Set `local` to `true` to stringify local address, otherwise `false` to stringify remote.

Set `ip` to `true` to stringify IP, `port` to stringify port. Example:

```
print('Connection from:', Net.ctos(conn, false, true, true));
```

Net.discard

```
Net.discard(conn, len)
```

Remove initial `len` bytes of data from the connection's `conn` receive buffer in order to discard that data and reclaim RAM to the system.

Net.serve

```
Net.serve(options)
```

Start TCP or UDP server. `options` is an object:

```
{
  // Required. Port to listen on, 'tcp://PORT' or `udp://PORT`.
  addr: 'tcp://1234',
  // Optional. Called when connection is established.
  onconnect: function(conn) {},
  // Optional. Called when new data is arrived.
  ondata: function(conn, data) {},
  // Optional. Called when protocol-specific event is triggered.
  onevent: function(conn, data, ev, edata) {},
  // Optional. Called when the connection is about to close.
  onclose: function(conn) {},
  // Optional. Called when on connection error.
  onerror: function(conn) {},
}
```

Example - a UDP echo server. Change `udp://` to `tcp://` to turn this example into the TCP echo server:

```
Net.serve({
  addr: 'udp://1234',
  ondata: function(conn, data) {
    print('Received from:', Net.ctos(conn, false, true, true), ':', data);
    Net.send(conn, data);           // Echo received data back
    Net.discard(conn, data.length); // Discard received data
  },
});
```

Net.connect

```
Net.connect(options)
```

Connect to a remote host. `options` is the same as for the `Net.serve`.

The `addr` format is `[PROTO://]HOST:PORT`. `PROTO` could be `tcp` or

`udp`. `HOST` could be an IP address or a host name. If `HOST` is a name, it will be resolved asynchronously.

Examples of valid addresses: `google.com:80`, `udp://1.2.3.4:53`, `10.0.0.1:443`, `[::1]:80`.

Net.close

```
Net.close(conn)
```

Send all pending data to the remote peer, and disconnect when all data is sent.

Return value: none.

Net.send

```
Net.send(conn, data)
```

Send data to the remote peer. `data` is an mJS string.

Return value: none.

Net.EVENT_GRP

```
Net.EVENT_GRP
```

Net events group, to be used with `Event.addGroupHandler()`. Possible events are:

- `Net.STATUS_DISCONNECTED`
- `Net.STATUS_CONNECTING`
- `Net.STATUS_CONNECTED`
- `Net.STATUS_GOT_IP`

UART

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_uart.h	mgos_uart.c	api_uart.js

See https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter for more information about UART.

mgos_uart_configure

```
bool mgos_uart_configure(int uart_no, const struct mgos_uart_config *cfg);
```

Apply given UART configuration.

Example:

```
int uart_no = 0;

struct mgos_uart_config ucfg;
mgos_uart_config_set_defaults(uart_no, &ucfg);

ucfg.baud_rate = 115200;
ucfg.rx_buf_size = 1500;
ucfg.tx_buf_size = 1500;

if (!mgos_uart_configure(uart_no, &ucfg)) {
    LOG(LL_ERROR, ("Failed to configure UART%d", uart_no));
}
```

mgos_uart_config_set_defaults

```
void mgos_uart_config_set_defaults(int uart_no, struct mgos_uart_config *cfg);
```

Fill provided `cfg` structure with the default values. See example above.

mgos_uart_config_get

```
bool mgos_uart_config_get(int uart_no, struct mgos_uart_config *cfg);
```

Fill provided `cfg` structure with the current UART config.

Returns false if the specified UART has not been configured yet.

(*mgos_uart_dispatcher_t)

```
typedef void (*mgos_uart_dispatcher_t)(int uart_no, void *arg);
```

UART dispatcher signature, see `mgos_uart_set_dispatcher()`

mgos_uart_set_dispatcher

```
void mgos_uart_set_dispatcher(int uart_no, mgos_uart_dispatcher_t cb,
                             void *arg);
```

Set UART dispatcher: a callback which gets called when there is data in the input buffer or space available in the output buffer.

mgos_uart_write

```
size_t mgos_uart_write(int uart_no, const void *buf, size_t len);
```

Write data to the UART.

Note: if there is enough space in the output buffer, the call will return immediately, otherwise it will wait for buffer to drain.

If you want the call to not block, check `mgos_uart_write_avail()` first.

mgos_uart_write_avail

```
size_t mgos_uart_write_avail(int uart_no);
```

Returns amount of space available in the output buffer.

mgos_uart_printf

```
int mgos_uart_printf(int uart_no, const char *fmt, ...);
```

Write data to UART, printf style.

Note: currently this requires that data is fully rendered in memory before sending. There is no fixed limit as heap allocation is used, but be careful when printing longer strings.

mgos_uart_read

```
size_t mgos_uart_read(int uart_no, void *buf, size_t len);
```

Read data from UART input buffer.

Note: unlike write, read will not block if there are not enough bytes in the input buffer.

mgos_uart_read_mbuf

```
size_t mgos_uart_read_mbuf(int uart_no, struct mbuf *mb, size_t len);
```

Like `mgos_uart_read`, but reads into an mbuf.

mgos_uart_read_avail

```
size_t mgos_uart_read_avail(int uart_no);
```

Returns the number of bytes available for reading.

mgos_uart_set_rx_enabled

```
void mgos_uart_set_rx_enabled(int uart_no, bool enabled);
```

Controls whether UART receiver is enabled.

mgos_uart_is_rx_enabled

```
bool mgos_uart_is_rx_enabled(int uart_no);
```

Returns whether UART receiver is enabled.

mgos_uart_flush

```
void mgos_uart_flush(int uart_no);
```

Flush the UART output buffer - waits for data to be sent.

mgos_uart_schedule_dispatcher

```
void mgos_uart_schedule_dispatcher(int uart_no, bool from_isr);
```

Schedule a call to dispatcher on the next `mongoose_poll`

mgos_uart_get_stats

```
const struct mgos_uart_stats *mgos_uart_get_stats(int uart_no);
```

Get UART statistics

JS API

UART.setConfig

```
UART.setConfig(uartNo, param)
```

Set UART config. `param` is an object with the following optional fields:

- `baudRate`: baud rate, integer, default: 115200;
- `numDataBits`: Number of data bits, default: 8;
- `parity`: Parity: 0 - none, 1 - even, 2 - odd; default: none;
- `numStopBits`: Number of stop bits: 1 - 1 bit, 2 - 2 bits, 3 - 1.5; default: 1;
- `rxBufSize`: size of the Rx buffer, integer, default: 256;
- `rxFlowControl`: whether Rx flow control (RTS pin) is enabled, boolean, default: false;
- `rxLingerMicros`: how many microseconds to linger after Rx fifo is empty, in case more data arrives. Integer, default: 15;
- `txBufSize`: size of the Tx buffer, integer, default: 256;
- `txFlowControl`: whether Tx flow control (CTS pin) is enabled, boolean, default: false;

Other than that, there are architecture-dependent settings, grouped in the objects named with the architecture name: "esp32", "esp8266", etc.

Settings for esp32:

```

esp32: {
/*
 * GPIO pin numbers, default values depend on UART.
 *
 * UART 0: Rx: 3, Tx: 1, CTS: 19, RTS: 22
 * UART 1: Rx: 13, Tx: 14, CTS: 15, RTS: 16
 * UART 2: Rx: 17, Tx: 25, CTS: 26, RTS: 27
 */
gpio: {
    rx: number,
    tx: number,
    cts: number,
    rts: number,
},
/* Hardware FIFO tweaks */
fifo: {
/*
 * A number of bytes in the hardware Rx fifo, should be between 1 and 127.
 * How full hardware Rx fifo should be before "rx fifo full" interrupt is
 * fired.
 */
rxFullThresh: number,
/*
 * A number of bytes in the hardware Rx fifo, should be more than
 * rx_fifo_full_thresh.
 *
 * How full hardware Rx fifo should be before CTS is deasserted, telling
 * the other side to stop sending data.
 */
rxFcThresh: number,
/*
 * Time in uart bit intervals when "rx fifo full" interrupt fires even if
 * it's not full enough
 */
rxAlarm: number,
/*
 * A number of bytes in the hardware Tx fifo, should be between 1 and 127.
 * When the number of bytes in Tx buffer becomes less than
 * tx_fifo_empty_thresh, "tx fifo empty" interrupt fires.
 */
txEmptyThresh: number,
},
}

```

UART.setDispatcher

```
UART.setDispatcher(uartNo, callback, userdata)
```

Set UART dispatcher

callback which gets invoked when there is a new data in the input buffer
or when the space becomes available on the output buffer.

Callback receives the following arguments: (uartNo, userdata).

UART.write

```
UART.write(uartNo, data)
```

Write data to the buffer. Returns number of bytes written.

Example usage: `UART.write(1, "foobar")`, in this case, 6 bytes will be written.

UART.writeAvail

```
UART.writeAvail(uartNo)
```

Return amount of space available in the output buffer.

UART.read

```
UART.read(uartNo)
```

It never blocks, and returns a string containing read data (which will be empty if there's no data available).

UART.readAvail

```
UART.readAvail(uartNo)
```

Return amount of data available in the input buffer.

UART.setRxEnabled

```
UART.setRxEnabled(uartNo)
```

Set whether Rx is enabled.

UART.isRxEnabled

```
UART.isRxEnabled(uartNo)
```

Returns whether Rx is enabled.

UART.flush

```
UART.flush(uartNo)
```

Flush the UART output buffer, wait for the data to be sent.

Config

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_sys_config.h	mgos_sys_config.c	api_config.js

A lot of the Mongoose OS functionality is driven by the device configuration.

For example, in order to make a device connected to the MQTT server, there is no need to write a single line of code. It is enough to modify `mqtt.*` configuration settings.

A configuration infrastructure is described in the user guide. Below is the programmatic API for the device configuration.

mgos_sys_config_save

```
bool mgos_sys_config_save(const struct mgos_config *cfg, bool try_once,
                           char **msg);
```

Save config. Performs diff against defaults and only saves diffs.

Reboot is required to reload the config.

If try_once is set, the config will only take effect for one boot.

If return value is false, a message may be provided in *msg.

If non-NULL, it must be free()'d.

It is safe to pass a NULL msg

mgos_sys_config_save_level

```
bool mgos_sys_config_save_level(const struct mgos_config *cfg,
                                 enum mgos_config_level level, bool try_once,
                                 char **msg);
```

Saves given config at the specified level. Performs diff against level-1.

mgos_sys_config_load_level

```
bool mgos_sys_config_load_level(struct mgos_config *cfg,
                                enum mgos_config_level level);
```

Loads config up to and including level.

save_cfg

```
bool save_cfg(const struct mgos_config *cfg, char **msg);
```

Deprecated API, equivalent to mgos_sys_config_save(cfg, false, msg).

load_config_defaults

```
bool load_config_defaults(struct mgos_config *cfg);
```

Loads configs up to MGOS_CONFIG_LEVEL_USER - 1. Deprecated.

mgos_config_reset

```
void mgos_config_reset(int level);
```

Reset config down to and including |level|.

0 - defaults, 1-8 - vendor levels, 9 - user.

mgos_config_reset(MGOS_CONFIG_LEVEL_USER) will wipe user settings.

(*mgos_config_validator_fn)

```
typedef bool (*mgos_config_validator_fn)(const struct mgos_config *cfg,
                                         char **msg);
void mgos_sys_config_register_validator(mgos_config_validator_fn fn);
```

Register a config validator.

Validators will be invoked before saving config and if any of them returns false, config will not be saved.

An error message may be *msg may be set to error message.

Note: if non-NULL, *msg will be freed. Remember to use strdup and asprintf.

mgos_config_validate

```
bool mgos_config_validate(const struct mgos_config *cfg, char **msg);
```

Run validators on the specified config.

mgos_expand_mac_address_placeholders

```
void mgos_expand_mac_address_placeholders(char *str);
```

Expands question marks in "str" with digits from the MAC address.

mgos_config_apply

```
bool mgos_config_apply(const char *sys_config_subset_json, bool save);
```

Apply a subset of system configuration. Return true on success.

mgos_config_apply_s

```
bool mgos_config_apply_s(const struct mg_str str, bool save);
```

Same as mgos_config_apply but uses mg_str

mgos_sys_config_parse_sub

```
bool mgos_sys_config_parse_sub(const struct mg_str json, const char *section,
                               void *cfg);
```

Parse a subsection of sys config, e.g. just "spi".

cfg must point to the subsection's struct.

Example:

```
struct mgos_config_spi cfg;
const struct mg_str json_cfg = MG_MK_STR("{\"unit_no\": 1}");
memset(&cfg, 0, sizeof(cfg));
mgos_sys_config_parse_sub(json_cfg, "spi", &cfg);
```

JS API

Cfg.get

```
Cfg.get(path)
```

Get the config value by the configuration variable. Currently, only simple types are returned: strings, ints, booleans, doubles. Objects are not yet supported.

Examples:

```
load('api_config.js');
Cfg.get('device.id');           // returns a string
Cfg.get('debug.level');         // returns an integer
Cfg.get('wifi.sta.enable');     // returns a boolean
```

Cfg.set

```
Cfg.set(obj, opt_save)
```

Set the configuration. `obj` must be a subset of the whole configuration tree. `save` is boolean flag that indicating whether the change should be saved - it could be omitted, in which case it defaults to `true`.

Examples:

```
load('api_config.js');
Cfg.set({wifi: {ap: {enable: false}}}); // Disable WiFi AP mode
Cfg.set({debug: {level: 3}}); // Set debug level to 3
```

mgos_iram.h

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_iram.h	mgos_iram.c	

_IRAM_STR_LIT

```
# define _IRAM_STR_LIT(a) #a
<p class="mume-header " id="define-_iram_str_lita-a"></p>

# define _IRAM_STR(a) _IRAM_STR_LIT(a)
<p class="mume-header " id="define-_iram_stra-_iram_str_lita"></p>
```

Provides the IRAM macro that creates unique sections under .text so that unused functions can be GC'd.

mgos_file_utils.h

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_file_utils.h	mgos_file_utils.c	

mgos_file_copy

```
bool mgos_file_copy(const char *from, const char *to);
```

Copy a file

mgos_file_digest

```
bool mgos_file_digest(const char *fname, mbedtls_md_type_t dt, uint8_t *digest);
```

Compute file's digest. `*digest` must have enough space for the digest type.

mgos_file_copy_if_different

```
bool mgos_file_copy_if_different(const char *from, const char *to);
```

Copy the file if target does not exist or is different.

GPIO

GitHub Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_gpio.h	mgos_gpio.c	api_gpio.js

(*mgos_gpio_int_handler_f)

```
typedef void (*mgos_gpio_int_handler_f)(int pin, void *arg);
```

GPIO interrupt handler signature.

mgos_gpio_set_mode

```
bool mgos_gpio_set_mode(int pin, enum mgos_gpio_mode mode);
```

Set mode - input or output.

mgos_gpio_set_pull

```
bool mgos_gpio_set_pull(int pin, enum mgos_gpio_pull_type pull);
```

Set pull-up or pull-down type.

mgos_gpio_setup_input

```
bool mgos_gpio_setup_input(int pin, enum mgos_gpio_pull_type pull);
```

Sets up a pin as an input and configures pull-up or pull-down.

mgos_gpio_setup_output

```
bool mgos_gpio_setup_output(int pin, bool level);
```

Sets up pin output while avoiding spurious transitions:
desired output level is configured first, then mode.

mgos_gpio_read

```
bool mgos_gpio_read(int pin);
```

Read pin input level.

mgos_gpio_write

```
void mgos_gpio_write(int pin, bool level);
```

Set pin's output level.

mgos_gpio_toggle

```
bool mgos_gpio_toggle(int pin);
```

Flip output pin value. Returns value that was written.

mgos_gpio_read_out

```
bool mgos_gpio_read_out(int pin);
```

Read the value of the output register.

mgos_gpio_set_int_handler

```
bool mgos_gpio_set_int_handler(int pin, enum mgos_gpio_int_mode mode,
                               mgos_gpio_int_handler_f cb, void *arg);
```

Install a GPIO interrupt handler.

This will invoke handler on the main task, which makes it possible to use any functions but may delay servicing of the interrupt. If lower latency is required, use `mgos_gpio_set_int_handler_isr`, but you'll need to understand the implications, which are platform-specific.

Interrupt is automatically cleared once upon triggering. Then it is disabled until the handler gets a chance to run, at which point it is re-enabled. At this point it may re-trigger immediately if the interrupt condition arose again while the handler was pending or running. Handler may use `mgos_gpio_clear_int` to explicitly clear the condition.

Note that this will not enable the interrupt, this must be done explicitly with `mgos_gpio_enable_int()`.

mgos_gpio_set_int_handler_isr

```
bool mgos_gpio_set_int_handler_isr(int pin, enum mgos_gpio_int_mode mode,
                                   mgos_gpio_int_handler_f cb, void *arg);
```

Same as `mgos_gpio_set_int_handler` but invokes handler in ISR context, without the overhead of a context switch. GPIO interrupts are disabled while the handler is running.

mgos_gpio_enable_int

```
bool mgos_gpio_enable_int(int pin);
```

Enable interrupt on the specified pin.

mgos_gpio_disable_int

```
bool mgos_gpio_disable_int(int pin);
```

Disables interrupt (without removing the handler).

mgos_gpio_clear_int

```
void mgos_gpio_clear_int(int pin);
```

Clears a GPIO interrupt flag.

mgos_gpio_remove_int_handler

```
void mgos_gpio_remove_int_handler(int pin, mgos_gpio_int_handler_f *old_cb,
                                   void **old_arg);
```

Removes a previously set interrupt handler.

If `cb` and `arg` are not NULL, they will contain previous handler and arg.

mgos_gpio_set_button_handler

```
bool mgos_gpio_set_button_handler(int pin, enum mgos_gpio_pull_type pull_type,
                                  enum mgos_gpio_int_mode int_mode,
                                  int debounce_ms, mgos_gpio_int_handler_f cb,
                                  void *arg);
```

Handle a button on the specified pin.

Configures the pin for input with specified pull-up and performs debouncing:
upon first triggering user's callback is invoked immediately but further
interrupts are inhibited for the following debounce_ms milliseconds.

Typically 50 ms of debouncing time is sufficient.

`int_mode` is one of the `MGOS_GPIO_INT_EDGE_*` values and will specify whether
the handler triggers when button is pressed, released or both.

Which is which depends on how the button is wired: if the normal state is
pull-up (typical), then `MGOS_GPIO_INT_EDGE_NEG` is press and
`_POS` is release.

Calling with `cb` = NULL will remove a previously installed handler.

Note: implicitly enables the interrupt.

mgos_gpio_blink

```
bool mgos_gpio_blink(int pin, int on_ms, int off_ms);
```

A utility function that takes care of blinking an LED.

The pin must be configured as output first.

On (output "1") and off ("0") times are specified in milliseconds.

Set to (0, 0) to disable.

mgos_gpio_str

```
const char *mgos_gpio_str(int pin_def, char buf[8]);
```

String representation of pin number.

Will return "PA5" or "PK3" for platforms that have port banks.

JS API

GPIO.set_mode

```
GPIO.set_mode(pin, mode)
```

Set GPIO pin mode.

`mode` can be either `GPIO.MODE_INPUT` or `GPIO.MODE_OUTPUT`.

GPIO.set_pull

```
GPIO.set_pull(pin, pull_type)
```

Set GPIO pin pull type.

`pull_type` can be either `GPIO.PULL_NONE`, `GPIO.PULL_UP`, or `GPIO.PULL_DOWN`.

GPIO.setup_input

```
GPIO.setup_input(pin, pull_type)
```

Setup pin as input and configure pull type.

`pull_type` can be either `GPIO.PULL_NONE`, `GPIO.PULL_UP`, or `GPIO.PULL_DOWN`.

GPIO.setup_output

```
GPIO.setup_output(pin, level)
```

Setup pin as output and set initial level, 0 or 1.

Avoids spurious transitions: applies level first, then sets mode.

GPIO.toggle

```
GPIO.toggle(pin)
```

Toggle the level of certain GPIO pin.

Return value: 0 or 1, indicating the resulting pin level.

GPIO.write

```
GPIO.write(pin, level)
```

Set GPIO pin level to either 0 or 1. Return value: none.

GPIO.read

```
GPIO.read(pin)
```

Read GPIO pin level. Return value: 0 or 1.

GPIO.enable_int

```
GPIO.enable_int(pin)
```

Enable interrupts on GPIO pin.

This function must be called AFTER the interrupt handler is installed.

Return value: 1 in case of success, 0 otherwise.

GPIO.disable_int

```
GPIO.disable_int(pin)
```

Disable interrupts on GPIO pin.

Return value: 1 in case of success, 0 otherwise.

GPIO.blink

```
GPIO.blink(pin, on_ms, off_ms)
```

A utility function that takes care of blinking an LED.

The pin must be configured as output first.

On (output "1") and off ("0") times are specified in milliseconds.

Set to (0, 0) to disable.

Return value: 1 on success, 0 on failure.

GPIO.set_int_handler

```
GPIO.set_int_handler(pin, mode, handler)
```

Install GPIO interrupt handler. `mode` could be one of: `GPIO.INT_NONE`, `GPIO.INT_EDGE_POS`, `GPIO.INT_EDGE_NEG`, `GPIO.INT_EDGE_ANY`, `GPIO.INT_LEVEL_HI`, `GPIO.INT_LEVEL_LO`.

Return value: 1 in case of success, 0 otherwise.

Example:

```
GPIO.set_mode(pin, GPIO.MODE_INPUT);
GPIO.set_int_handler(pin, GPIO.INT_EDGE_NEG, function(pin) {
  print('Pin', pin, 'got interrupt');
}, null);
GPIO.enable_int(pin);
```

GPIO.set_button_handler

```
GPIO.set_button_handler(pin, pull, intmode, period, handler)
```

Install

GPIO button handler. `pull` is pull type, `intmode` is interrupt mode, `period` is debounce interval in milliseconds, `handler` is a function that receives pin number.

Return value: 1 in case of success, 0 otherwise.

Example:

```
GPIO.set_button_handler(pin, GPIO.PULL_UP, GPIO.INT_EDGE_NEG, 200, function(x) {
  print('Button press, pin: ', x);
}, null);
```

- [App](#)
- [Bitbang](#)
- [Debug \(UART\)](#)
- [Event](#)
- [mgos_file_utils.h](#)
- [GPIO](#)
- [mgos_iram.h](#)
- [Net events](#)
- [Config](#)
- [System](#)
- [Time](#)
- [Timers](#)
- [UART](#)
- [Utils](#)
- [JSON](#)

Utils

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_utils.h	mgos_utils.c	

mgos_system_restart_after

```
void mgos_system_restart_after(int delay_ms);
```

Restart system after the specified number of milliseconds

mgos_rand_range

```
float mgos_rand_range(float from, float to);
```

Return random number in a given range.

Event

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_event.h	mgos_event.c	api_events.js

Mongoose OS provides a way to get a notification when certain event happens. Each event has an associated event data passed as `void *`.

MGOS_EVENT_BASE

```
# define MGOS_EVENT_BASE(a, b, c) ((a) << 24 | (b) << 16 | (c) << 8)
<p class="mume-header " id="define-mgos_event_base-a-b-c-a-24-b-16-c-8"></p>
```

Macro to generate unique base event number.

A library can use the last byte (LSB) to create up to 256 unique events (see enum below on how Mongoose OS core defines its events).

A library should call `mgos_event_register_base()` in order to claim it and prevent event number conflicts.

MGOS_EVENT_BASE

```
# define MGOS_EVENT_SYS MGOS_EVENT_BASE('M', 'O', 'S')
<p class="mume-header " id="define-mgos_event_sys-mgos_event_base-m-o-s"></p>
```

These events are registered by the MGOS core.

Other events could be registered by the external libraries.

mgos_event_register_base

```
bool mgos_event_register_base(int base_event_number, const char *name);
```

Register a base event number in order to prevent event number conflicts.

Use `MGOS_EVENT_BASE()` macro to get `base_event_number`; `name` is an arbitrary name of the module who registers the base number.

Example:

```
#define MY_EVENT_BASE MGOS_EVENT_BASE('F', '0', '0')

enum my_event {
    MY_EVENT_AAA = MY_EVENT_BASE,
    MY_EVENT_BBB,
    MY_EVENT_CCC,
};

// And somewhere else:
mgos_event_register_base(MY_EVENT_BASE, "my module foo");
```

mgos_event_trigger

```
int mgos_event_trigger(int ev, void *ev_data);
```

Trigger an event `ev` with the event data `ev_data`. Return number of event handlers invoked.

(*mgos_event_handler_t)

```
typedef void (*mgos_event_handler_t)(int ev, void *ev_data, void *userdata);
```

Event handler signature.

mgos_event_add_handler

```
bool mgos_event_add_handler(int ev, mgos_event_handler_t cb, void *userdata);
```

Add an event handler. Return true on success, false on error (e.g. OOM).

Example:

```
static void system_restart_cb(int ev, void *ev_data, void *userdata) {
    LOG(LL_INFO, ("Going to reboot!"));
    (void) ev;
    (void) ev_data;
    (void) userdata;
}

// And somewhere else:
mgos_event_add_handler(MGOS_EVENT_REBOOT, system_restart_cb, NULL);
```

mgos_event_add_group_handler

```
bool mgos_event_add_group_handler(int evgrp, mgos_event_handler_t cb,
                                  void *userdata);
```

Like `mgos_event_add_handler()`, but subscribes on all events in the given group `evgrp`. Event group includes all events from `evgrp & ~0xff` to `evgrp | 0xff`.

Example:

```
static void sys_event_cb(int ev, void *ev_data, void *userdata) {
    LOG(LL_INFO, ("Got system event %d", ev));
    (void) ev;
    (void) ev_data;
    (void) userdata;
}

// And somewhere else:
mgos_event_add_handler(MGOS_EVENT_SYS, sys_event_cb, NULL);
```

mgos_event_remove_handler

```
bool mgos_event_remove_handler(int ev, mgos_event_handler_t cb, void *userdata);
bool mgos_event_remove_group_handler(int evgrp, mgos_event_handler_t cb,
                                     void *userdata);
```

Remove an event handler.

Both cb and userdata must match the initial add invocation.

Returns true if a handler was found and removed, false if there was no such handler in the first place.

JS API

Event.addHandler

```
Event.addHandler(ev, callback, userdata)
```

Add a handler for the given event `ev`. Callback should look like:

```
function(ev, evdata, userdata) { /* ... */ }
```

Example:

```
Event.addHandler(Event.REBOOT, function(ev, evdata, ud) {
    print("Going to reboot!");
}, null);
```

Event.addGroupHandler

```
Event.addGroupHandler(evgrp, callback, userdata)
```

Like `Event.addHandler()`, but subscribes on all events in the given event group `evgrp`. Event group includes all events from `evgrp & ~0xff` to `evgrp | 0xff`.

Example:

```
Event.addGroupHandler(Event.SYS, function(ev, evdata, ud) {
    print("Sys event:", ev);
}, null);
```

Event.on

```
Event.on(event_num, callback, userdata)
```

Alias for `Event.addHandler`

Event.regBase

```
Event.regBase(base_event_number, name)
```

Register a base event number in order to prevent event number conflicts.

Use `Event.baseNumber(id)` to get `base_event_number`; `name` is an arbitrary event name.

Example:

```
let bn = Event.baseNumber("ABC");
if (!Event.regBase(bn, "My module")) {
  die("Failed to register base event number");
}

let MY_EVENT_FOO = bn + 0;
let MY_EVENT_BAR = bn + 1;
let MY_EVENT_BAZ = bn + 2;
```

Event.baseNumber

```
Event.baseNumber(id)
```

Generates unique base event number (32-bit) by a 3-char string.

LSB is always zero, and a library can use it to create up to 256 unique events.

A library should call `Event.regBase()` in order to claim it and prevent event number conflicts. (see example there)

Event.trigger

```
Event.trigger(ev, evdata)
```

Trigger an event with the given id `ev` and event data `evdata`.

Event.evdataLogStr

```
Event.evdataLogStr(evdata)
```

Getter function for the `evdata` given to the event callback for the event

`Event.LOG`, see `Event.addHandler()`.

Event.LOG

```
Event.LOG
```

System event which is triggered every time something is printed to the log. In the callback, use `Event.evdataLogStr(evdata)` to get string which was printed.

Event.REBOOT

```
Event.REBOOT
```

System event which is triggered right before going to reboot. `evdata` is irrelevant for this event.

Event.OTA_STATUS

Event.OTA_STATUS

System event which is triggered when OTA status changes.

In the callback, use `OTA.evdataOtaStatusMsg(evdata)` from `api_ota.js` to get the OTA status message.

Event.CLOUD_CONNECTED

Event.CLOUD_CONNECTED

Triggered when device is connected to the cloud (mqtt, dash)

Event.CLOUD_DISCONNECTED

Event.CLOUD_DISCONNECTED

Triggered when device is disconnected from the cloud

Timers

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_timers.h	mgos_timers.c	api_timer.js

Mongoose OS supports two types of timers: software timers and hardware timers.

- Software timers. Implemented as Mongoose library events, in software.
Timer callback is called in a Mongoose task context. Frequency is specified in milliseconds. Number of software timers is not limited.
Timer intervals cannot be short - limited by the underlying task scheduling. For example, if you want a very frequent sensor reading, like thousand readings a second, use hardware timer instead.
Both C and JavaScript API is available.
- Hardware timers. Implemented in hardware. Timer callback is executed in the ISR context, therefore it can do a limited set of actions.
Number of hardware timers is limited: (ESP8266: 1, ESP32: 4, CC32xx: 4).
Frequency is specified in microseconds. Only C API is present, because calling to JS requires switching to Mongoose task context.

Usage example:

```

# include "mgos_app.h"
<p class="mume-header" id="include-mgos_apph"></p>

# include "mgos_system.h"
<p class="mume-header" id="include-mgos_systemh"></p>

# include "mgos_timers.h"
<p class="mume-header" id="include-mgos_timersh"></p>

static void my_timer_cb(void *arg) {
    bool val = mgos_gpio_toggle(mgos_sys_config_get_pins_led());
    LOG(LL_INFO, ("uptime: %.2lf", mgos_uptime()));
    (void) arg;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_set_timer(1000, MGOS_TIMER_REPEAT, my_timer_cb, NULL);
    return MGOS_APP_INIT_SUCCESS;
}

```

(*timer_callback)

```
typedef void (*timer_callback)(void *param);
```

Timer callback

mgos_set_timer

```
mgos_timer_id mgos_set_timer(int msecs, int flags, timer_callback cb,
                             void *cb_arg);
```

Setup a timer with `msecs` timeout and `cb` as a callback.

`flags` is a bitmask, currently there's only one flag available:

`MGOS_TIMER_REPEAT` (see above). `arg` is a parameter to pass to `cb`.

Return numeric timer ID.

Note that this is a software timer, with fairly low accuracy and high jitter.

However, number of software timers is not limited.

If you need intervals < 10ms, use `mgos_set_hw_timer`.

Example:

```

static void my_timer_cb(void *arg) {
    bool val = mgos_gpio_toggle(mgos_sys_config_get_pins_led());
    LOG(LL_INFO, ("uptime: %.2lf", mgos_uptime()));
    (void) arg;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_set_timer(1000, MGOS_TIMER_REPEAT, my_timer_cb, NULL);
    return MGOS_APP_INIT_SUCCESS;
}

```

mgos_set_hw_timer

```
mgos_timer_id mgos_set_hw_timer(int usecs, int flags, timer_callback cb,
                                void *cb_arg);
```

Setup a hardware timer with `usecs` timeout and `cb` as a callback.

This is similar to `mgos_set_timer`, but can be used for shorter intervals (note that time unit is microseconds).

Number of hardware timers is limited (ESP8266: 1, ESP32: 4, CC32xx: 4).

Callback is executed in ISR context, with all the implications of that.

mgos_clear_timer

```
void mgos_clear_timer(mgos_timer_id id);
```

Disable timer with a given timer ID.

mgos_uptime

```
double mgos_uptime(void);
```

Get number of seconds since last reboot

mgos_strerror

```
int mgos_strerror(char *s, int size, char *fmt, int time);
```

Format `time` according to a `strftime()`-conformant format.

Write the result into the `s,size` buffer. Return resulting string length.

JS API

Timer.set

```
Timer.set(milliseconds, flags, handler, userdata)
```

Setup timer with `milliseconds` timeout and `handler` as a callback.

`flags` can be either 0 or `Timer.REPEAT`. In the latter case, the call will be repeated indefinitely (but can be cancelled with `Timer.del()`), otherwise it's a one-off.

Return value: numeric timer ID.

Example:

```
// Call every second
Timer.set(1000, Timer.REPEAT, function() {
  let value = GPIO.toggle(2);
  print(value ? 'Tick' : 'Tock');
}, null);
```

Timer.now

```
Timer.now()
```

Return current time as double value, UNIX epoch (seconds since 1970).

Timer.del

```
Timer.del(id)
```

Cancel previously installed timer.

Timer(fmt)

```
Timer.fmt(fmt, time)
```

Formats the time 'time' according to the strftime-like format specification 'fmt'. The strftime reference can be found e.g.

[here](#).

Example:

```
let now = Timer.now();
let s = Timer.fmt("Now it's %I:%M%p.", now);
print(s); // Example output: "Now it's 12:01AM."
```

App

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_app.h	mgos_app.c	

Definitions for the user-defined app entry point, `mgos_app_init()`.

The `mgos_app_init()` function is like the `main()` function in the C program. This is a app's entry point.

The mongoose-os core code does implement `mgos_app_init()` stub function as a weak symbol, so if user app does not define its own `mgos_app_init()`, a default stub will be used. That's what most of the JavaScript based apps do - they do not contain C code at all.

mgos_app_init

```
enum mgos_app_init_result mgos_app_init(void);
```

User app init function.

A weak stub is provided in `mgos_app_init.c`, which can be overridden.

Example of a user-defined init function:

```
#include "mgos_app.h"

enum mgos_app_init_result mgos_app_init(void) {
    if (!my_super_duper.hardware_init()) {
        LOG(LL_ERROR, ("something went bad"));
        return MGOS_APP_INIT_ERROR;
    }
    LOG(LL_INFO, ("my app initialised"));
    return MGOS_APP_INIT_SUCCESS;
}
```

mgos_app_preinit

```
void mgos_app_preinit(void);
```

An early init hook, for apps that want to take control early

in the init process. How early? very, very early. If the platform uses RTOS, it is not running yet. Dynamic memory allocation is not safe. Networking is not running. The only safe thing to do is to communicate to mg_app_init something via global variables or shut down the processor and go (back) to sleep.

Debug (UART)

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_debug.h	mgos_debug.c	api_log.js

mgos_debug_write

```
void mgos_debug_write(int fd, const void *buf, size_t len);
```

Write debug info buf, len to the given file descriptor fd.

mgos_debug_flush

```
void mgos_debug_flush(void);
```

Flush debug UARTs, both stdout and stderr.

mgos_set_stdout_uart

```
bool mgos_set_stdout_uart(int uart_no);
```

Set UART for stdout. Negative value disables stdout.

mgos_set_stderr_uart

```
bool mgos_set_stderr_uart(int uart_no);
```

Set UART for stderr. Negative value disables stderr.

mgos_get_stdout_uart

```
int mgos_get_stdout_uart(void);
```

Get stdout UART number; -1 indicates that stdout is disabled.

mgos_get_stderr_uart

```
int mgos_get_stderr_uart(void);
```

Get stderr UART number; -1 indicates that stderr is disabled.

mgos_debug_suspend_uart

```
void mgos_debug_suspend_uart(void);
```

Suspend UART output (both stdout and stderr); see [mgos_debug_resume_uart\(\)](#). Nested suspension is supported: UART needs to be resumed as many times as it was suspended.

mgos_debug_resume_uart

```
void mgos_debug_resume_uart(void);
```

Resume previously suspended UART output, see `mgos_debug_suspend_uart()`.

mgos_debug_uart_is_suspended

```
bool mgos_debug_uart_is_suspended(void);
```

Returns whether UART output is suspended at the moment.

JS API

Log.print

```
Log.print(level, msg)
```

Print message to stderr if provided

level is \geq `Cfg.get('debug.level')`. Possible levels are:

- `Log.ERROR` (0)
- `Log.WARN` (1)
- `Log.INFO` (2)
- `Log.DEBUG` (3)
- `Log.VERBOSE_DEBUG` (4)

Log.error

```
Log.error(msg)
```

Shortcut for `Log.print(Log.ERROR, msg)`

Log.warn

```
Log.warn(msg)
```

Shortcut for `Log.print(Log.WARN, msg)`

Log.info

```
Log.info(msg)
```

Shortcut for `Log.print(Log.INFO, msg)`

Log.debug

```
Log.debug(msg)
```

Shortcut for `Log.print(Log.DEBUG, msg)`

Log.verboseDebug

```
Log.verboseDebug(msg)
```

Bitbang

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_bitbang.h	mgos_bitbang.c	api_bitbang.js

Bit banging is a technique to implement hardware protocols fully in software, by managing microcontroller pin states - including timing, voltage levels, etc.

See [wikipedia article](#) for a detailed information.

Usage example:

```
# include "mgos_bitbang.h"
<p class="mume-header" id="include-mgos_bitbang"></p>

# include "mgos_gpio.h"
<p class="mume-header" id="include-mgos_gpioh"></p>

# include "mgos_system.h"
<p class="mume-header" id="include-mgos_systemh-1"></p>

void mgos_neopixel_show(struct mgos_neopixel *np) {
    mgos_gpio_write(np->pin, 0);
    mgos_usleep(60);
    mgos_bitbang_write_bits(np->pin, MGOS_DELAY_100NSEC, 3, 8, 8, 6, np->data,
                           np->num_pixels * NUM_CHANNELS);
    mgos_gpio_write(np->pin, 0);
    mgos_usleep(60);
    mgos_gpio_write(np->pin, 1);
}
```

mgos_bitbang_write_bits

```
void mgos_bitbang_write_bits(int gpio, enum mgos_delay_unit delay_unit, int t0h,
                             int t0l, int t1h, int t1l, const uint8_t *data,
                             size_t len);
```

Bit bang GPIO pin `gpio`. `len` bytes from `data` are sent to the specified pin bit by bit. Sending each bit consists of a "high" and "low" phases, length of which is determined by the specified timing parameters.

```
+---+
|   |
-+   +---+
      tXh  tXI
```

`t0h` and `t0l` specify timings if the bit being transmitted is 0, `t1h` and `t1l` specify the same for the case where the bit is 1. If any of these is < 0, the corresponding phase is skipped.

mgos_bitbang_write_bits_js

```
void mgos_bitbang_write_bits_js(int gpio, enum mgos_delay_unit delay_unit,
                                uint32_t t, const uint8_t *data, size_t len);
```

This function is a wrapper for `mgos_bitbang_write_bits()`. It has smaller number of arguments (less than 6) and therefore could be FFI-ed to JavaScript. Essentially, it just packs all time patterns into a single value `t`.

JS API

BitBang.write

```
BitBang.write(pin, delay_unit, t0h, t0l, t1h, t1l, ptr, len)
```

Write bits to a given pin. `delay_unit` is one of the:

`BitBang.DELAY_MSEC`, `BitBang.DELAY_USEC`, `BitBang.DELAY_100NSEC`.

`ptr`, `len` is a bit pattern to write. `t0h`, `t0l` is the time pattern

for zero bit, `t1h`, `t1l` is the time pattern for 1. The time pattern

specifies the number of time units to hold the pin high, and the number

of units to hold the pin low. Return value: none.

System

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_system.h	mgos_system.c	api_sys.js

HAL that needs to be implemented for each hardware platform.

mgos_get_heap_size

```
size_t mgos_get_heap_size(void);
```

Get system memory size.

mgos_get_free_heap_size

```
size_t mgos_get_free_heap_size(void);
```

Get system free memory.

mgos_get_min_free_heap_size

```
size_t mgos_get_min_free_heap_size(void);
```

Get minimal watermark of the system free memory.

mgos_get_fs_memory_usage

```
size_t mgos_get_fs_memory_usage(void);
```

Get filesystem memory usage

mgos_get_fs_size

```
size_t mgos_get_fs_size(void);
```

Get filesystem size.

Deprecated, please use mgos_vfs_get_space_total("/") instead.

mgos_get_free_fs_size

```
size_t mgos_get_free_fs_size(void);
```

Get filesystem free space.

Deprecated, please use mgos_vfs_get_space_free("/") instead.

mgos_fs_gc

```
void mgos_fs_gc(void);
```

Garbage-collect filesystem

mgos_wdt_feed

```
void mgos_wdt_feed(void);
```

Feed watchdog

mgos_wdt_set_timeout

```
void mgos_wdt_set_timeout(int secs);
```

Set watchdog timeout

mgos_wdt_enable

```
void mgos_wdt_enable(void);
```

Enable watchdog

mgos_wdt_disable

```
void mgos_wdt_disable(void);
```

Disable watchdog

mgos_system_restart

```
void mgos_system_restart(void) __attribute__((noreturn));
```

Restart system

mgos_msleep

```
void mgos_msleep(uint32_t msecs);
```

Delay given number of milliseconds

mgos_usleep

```
void mgos_usleep(uint32_t usecs);
```

Delay given number of microseconds

mgos_ints_disable

```
void mgos_ints_disable(void);
```

Disable interrupts

mgos_ints_enable

```
void mgos_ints_enable(void);
```

Enable interrupts

(*mgos_cb_t)

```
typedef void (*mgos_cb_t)(void *arg);
```

Callback for `mgos_invoke_cb()`

mgos_invoke_cb

```
bool mgos_invoke_cb(mgos_cb_t cb, void *arg, bool from_isr);
```

Invoke a callback in the main MGOS event loop.

Returns true if the callback has been scheduled for execution.

mgos_get_cpu_freq

```
uint32_t mgos_get_cpu_freq(void);
```

Get the CPU frequency in Hz

mgos_rlock_create

```
struct mgos_rlock_type *mgos_rlock_create(void);
```

Create a recursive lock.

mgos_rlock

```
void mgos_rlock(struct mgos_rlock_type *l);
```

Acquire a recursive lock.

mgos_unlock

```
void mgos_unlock(struct mgos_rlock_type *l);
```

Release a recursive lock.

mgos_rlock_destroy

```
void mgos_rlock_destroy(struct mgos_rlock_type *l);
```

Destroy a recursive lock.

JS API

Sys._sbuf

```
Sys._sbuf(len)
```

Helper function to allocate string of at least given length. Note that the resulting string is usually bigger than this, and it is always longer than 5 bytes; that's to guarantee that the string data is stored in a common buffer and not inlined into mjs_val_t, thus the buffer can be used as an "output" buffer: a string can be passed to some function which will alter the contents, and these changes will be visible to the caller.

Sys.calloc

```
Sys.calloc(nmemb, size)
```

Allocate a memory region.

Note: currently memory allocated this way must be explicitly released with free().

Sys.total_ram

```
Sys.total_ram()
```

Return total available RAM in bytes.

Sys.free_ram

```
Sys.free_ram()
```

Return free available RAM in bytes.

Sys.reboot

```
Sys.reboot(ms)
```

Reboot the system after ms milliseconds. Return value: none.

Sys.uptime

```
Sys.uptime()
```

Return number of seconds since last reboot.

Sys.usleep

```
Sys.usleep(microseconds)
```

Sleep given number of microseconds.

Return value: none.

Sys.wdt_feed

```
Sys.wdt_feed()
```

Feed the watchdog timer.

Return value: none.

CCM (Cloud Communication Module)

See <https://mdash.net/docs/quickstart/ccm.md>

Licensing

- Login to the [Mongoose License Manager](#)
- Buy any number of "ccm" licenses you need. IMPORTANT: choose "ccm" license type:
- If your device is connected to your workstation over USB, run:

```
mos license
```

- Or, if your device is accessible remotely, run:

```
mos --port DEVICE_ADDRESS license
```

Licenses are bound to devices, and they are permanent.

The `mos license` command creates a unique license string specific for a device,

and updates `device.license` configuration parameter, which "unlocks" the device and removes restrictions. The License Manager decreases the number of available licenses and saves a unique license string. The next time `mos license` is run for the same device, an old license string is returned by the License Manager but the number of available licenses is not decreased.

Thus a device, once licensed, is licensed permanently.

- [User guide](#)
- [Licensing](#)

REST API Reference

Every API call must be authenticated by sending `Authorization: Bearer KEY` HTTP header. Those API calls that use `POST` or `PUT`, should specify `application/json` mime type. Example - calling device's RPC function:

```
curl -H 'Content-Type: application/json' \
-H 'Authorization: Bearer API_KEY' \
-d '{"pin": 2}' \
https://dash.mongoose-os.com/api/v2/devices/DEVICE_ID/rpc/GPIO.Toggle
true
```

GET /devices

List all registered devices. Params: none.

POST /devices

Register new device. Params: none.

POST /devices/:id

Change device properties. Params:

```
{  
  "name": "MyCoolDeviceName",  
  "shared_with": "github_user1,github_user2",  
  "shadow": ...  
}
```

Any key in the params is optional.

DELETE /devices/:id

Delete device. Params: none.

POST /devices/:id/rpc/:func

Call device's RPC function. Params: any valid JSON string, which is expected by the function.

POST /devices/:id/ota

Perform device OTA. Params: a binary content of the firmware .zip file. Example: curl -v -F file=@fw.zip URL.

GET /devices/:id/data

Get device saved data. Params: none.

GET /keys

List all API keys. Params: none.

POST /keys

Create an API key. Params: none.

DELETE /keys/:id

Delete an API key. Params: none.

POST /logs

Get stored notification logs. Params: {"start": 0} - an offset to start from.

GET /devices/data

Get data from all devices. Params: none.

Device Management Dashboard



Mongoose OS provides an integrated service for device management called mDash.

It is located at <https://dash.mongooselabs.com/>, and provides the following

functionality:

- 24/7 access and monitoring for your devices in the field
- Full isolation. Other users cannot see traffic from your devices and cannot access them
- OTA updates, on-the-fly file editing, config editing, shadow editing, calling any device function like `Sys.Reboot`, `Sys.GetInfo`, `GPIO.Toggle`, etc
- RESTful API and Web UI that provide:
 - Online/offline status together with device metadata like firmware version, build time, device architecture
 - Full access to the devices' RPC services (remote management)
 - Reliable OTA updates with manual or automatic commit
 - Device twin (or, shadow) cloud object, semantically identical to AWS / Azure
- Notification stream that lets you catch state changes, implement custom logging, etc

Device registration

Step 1. Complete steps 1-7 [quickstart guide](#)

Step 2. Login to the <https://dash.mongooselabs.com/>, register a new device.

Left Click on a "Token" link to copy it to the clipboard.

Step 3. Configure device:

```
mos config-set dash.enable=true dash.token=TOKEN
```

Done! Now your device should appear "green" (online) on a dashboard.

Troubleshooting

If `mos config-set` command on Step 5 fails for you, complaining about non-existing variables, then your app needs to be rebuilt with the `dash` library. Edit your app's `mos.yml`:

```
libs:  
  ...  
  - origin: https://github.com/mongooselabs/dash # <-- Add this line!
```

Then, `mos build`, `mos flash`, and repeat registration from Step 3.

Using device simulator

If you don't want to use a real device for some reason, there is a POSIX program for Linux/Mac/Windows that could be used to simulate the device. Clone it from <https://github.com/cesanta/mongoose-os-device-simulator>.

Then simply run `make`. You'll see the access token prompt:

```
$ make
cc -W -Wall -g -O2 -D MG_ENABLE_CALLBACK_USERDATA=1 -D MG_ENABLE_SSL -DMG_SSL_IF=MG_SSL_IFMBEDTLS -Imbedtls -Imbedcrypto -Imbedx509 mai
./simulator
Enter access token:
```

Login to dash.mongoose-os.com, choose some registered device, copy the access token to the clipboard, paste into the prompt and press enter.

Licensing and Contact

mDash is distributed by Mongoose OS team as a standalone docker image that could be run on-premises, as well as a hosted service <https://dash.mongoose-os.com>.

In order to run mDash on-premises, install [Docker](#) and execute `docker run mgos/dash`. Customisation can be done via the configuration. By default, there is a restriction on the maximum number of users (5 maximum), with maximum of 2 devices per user.

Please [contact us](#) if you would like to use the mDash in your production setup with increased limits.

- [Introduction](#)
- [Device registration](#)
- [Technical overview](#)
- [OTA support](#)
- [Device shadow](#)
- [Notifications](#)
- [Device sharing](#)
- [Mobile app](#)
- [REST API reference](#)

Technical overview

A configured device connects to the dashboard over the secure Websocket, sending a `Authorization: Bearer GENERATED_TOKEN` handshake header. After the successful handshake, all communication is done via the JSON-RPC 2.0 protocol. Each RPC frame wrapped into the Websocket frame.

The RESTful API endpoints that the dashboard exports start with `/api/v2/`. The endpoints that are for the individual device start with `/api/v2/devices/:id`, where `:id` is the unique device ID, generated at the device registration together with the access token.

The dashboard gives a RESTful access to all RPC services exported by the device.

Thus, the dashboard acts as a JSON-RPC / RESTful bridge. The device's RPC methods are mapped to `/api/v2/devices/:id/rpc/:method` endpoints. If the RPC endpoint does not accept any parameters, you can use `GET` HTTP method. If it does, `POST` must be used. The parameters should be a JSON string, and the `Content-Type: application/json` header must be set, e.g.:

```
curl \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer API_KEY' \
-d '{"pin": 2}' \
http://dash.mongoose-os.com/api/v2/devices/DEVICE_ID/rpc/GPIO.Toggle
```

When a device comes online, the `dash` library that is responsible for the dashboard connection, sends a shadow update message with the information about the device: architecture, firmware version, etc:

Notifications

The dashboard provides a special secure Websocket endpoint `wss://dash.mongoose-os.com/api/v2/notify`. This is a read-only notifications endpoint.

Each notification is a JSON object with three keys:

- `name`: notification name, e.g. "online", "offline", "rpc.in.GetInfo", "rpc.out.Log"
- `id`: an ID of a device that generated the event
- `data`: optional notification-specific data.

The `online` and `offline` events are generated by the dashboard. The `rpc.out.*` events are generated by the device: these are JSON-RPC requests without an ID (notifications). For example, `dash` library forwards all device logs to the dashboard as `Log` RPC calls, thus generating `rpc.out.Log` events. RPC call to the device generate `rpc.in.*` events.

The dashboard UI uses `/api/v2/notify` endpoint in order to catch state changes. Login to the dashboard and open the developer tools / network / WS panel to see it in action.

How to catch notifications from all devices

You can implement your own service that attaches to the `/api/v2/notify`, for example in Node JS (don't forget to substitute `API_TOKEN` with your real API access token):

```
const Websocket = require('ws');
const addr = 'wss://dash.mongoose-os.com/api/v2/notify?access_token=API_TOKEN';
const ws = new Websocket(addr, { origin: addr });
ws.on('message', msg => console.log('Got message:', msg.toString()));
```

To see your mDash API tokens, login to mDash and click on "Keys" tab:

How to send a notification from your device

The API is documented at [mDash API](#).

Below are quick examples.

In C/C++:

```
mgos_dash_notify("MyStat", "{temperature: %f}", 12.34);
```

In JavaScript:

```
Dash.notify('MyStat', {temperature: 12.34});
```

Note that the format of data is arbitrary - it could be anything: a number, a string, an array, or a complex nested object. It's up to you. In this example, a simple object {"temperature": 12.34} is used. The example generates the following notification:

```
{
  "id": "DEVICE_ID",           // This is the device ID that generated notification
  "name": "rpc.out.MyStat",    // "rpc.out.EVENT_NAME", in our case rpc.out.MyStat
  "data": {"temperature": 12.34} // Your event payload - arbitrary
}
```

Further possible actions:

- Catch this notification (see section above) and do your custom action
- Modify the PWA mobile app, which catches all notifications, and show the data somehow to the user
- Save the data automatically to the database, and graph it. See [Data Storage](#) section

Device sharing

It is possible to share a device with other users. This is driven by the `shared_with` property of the device, which is a comma-separated string of user IDs to share device with. The `shared_with` property could be set via the API, or via the dashboard's "Info" panel:

Shared devices are marked with a user icon in the device list:

In order to see your user ID, click on the Account tab: # Mobile app

For every registered user, mDash serves custom Progressive Web App (PWA).

This gives an ability to manage user's devices via a mobile phone.

In order to enable PWA,

- Choose "Mobile App" in the menu
- Tick the "Enable mobile app" checkbox
- Enter your email address and press on "Send invitation" button

That creates a built-in mobile app for the whole fleet of devices.

Customizing built-in app

The default app is customisable. It displays a list of devices shared with the particular mobile app customer, and for each device, displays a list of control widgets. The information displayed by each widget is based on a device shadow. Read-only widgets just display certain shadow values, and "actuator" widgets (switches, buttons) modify shadow values. All device changes are caught via the websocket notifications, and displayed real-time.

The look and feel of the built-in app could be customised in by changing

the `var ui = { ... }`; object in the PWA code. This is the default value:

```
var ui = {
  title: 'My devices',
  // icon: 'fa-save',
  controls: [
    { path: 'on', type: 'toggle', title: 'LED on/off:' },
    { path: 'uptime', type: 'value', format: 'up: %D', icon: 'fa-clock-o' },
  ],
};
```

As you see, you could modify app's title, menu icon, and define an arbitrary number of control widgets. The description of the available widgets is below.

Toggle button

```
{ path: 'SHADOW_KEY', type: 'toggle', title: 'YOUR TITLE' }
```

Renders a toggle button with a label. A shadow key points to a boolean true/false.

Example:

```
var ui = {
  title: 'Cool app',
  controls: [
    { path: 'on', type: 'toggle', title: 'my light:' },
  ],
};
```

Value

```
{ path: 'SHADOW_KEY', type: 'value', icon: 'fa-save', format: '', title: 'hello' }
```

Renders shadow value. icon is optional, could be a font-awesome class, e.g. fa-save, or an inline base64-encoded icon data. The format could be:

- %D expects a number of seconds, converts it to a Xd Xh Xm Xs.
- %.Nf formats floating-point value, e.g. %.1f leaves one digits after comma.

Example:

```
var ui = {
  title: 'Cool app',
  controls: [
    { path: 'on', type: 'toggle', title: 'my light:' },
    { path: 'uptime', type: 'value', title: 'up:', icon: 'fa-clock-o', format: 'up: %D' },
  ],
};
```

Push button

```
{ path: 'SHADOW_KEY', type: 'button', icon: 'fa-save', title: 'hello' }
```

Renders a push button. icon is optional, could be a font-awesome class, e.g. fa-save, or an inline base64-encoded icon data. The shadow value must be an numeric value. Pushing a button will increment it and set incremented value in desired.

Example:

```

var ui = {
  title: 'Cool app',
  controls: [
    { path: 'on', type: 'toggle', title: 'my light:' },
    { path: 'uptime', type: 'value', title: 'up:', icon: 'fa-clock-o', format: 'up: %D' },
    { path: 'v1', type: 'button', title: 'start action', icon: 'fa-flash' },
  ],
};

```

Fully custom app

You can save your own code and totally modify the look and feel of the app.

If you do so, you can always restore the built-in app by pressing the

"Restore default app" button.

Mobile app API

mDash exports a different set of RESTful API to the mobile app. Mobile app

API allows to get a list of devices, and render the UI according to their shadows.

Method	Endpoint	Params	Description
GET	/api/v2/appuser		Get list of devices
POST	/api/v2/appuser/device/:id	{shadow}	Update device shadow
WSS	/api/v2/appuser/ws		WebSocket notification URI

Device shadow

Dashboard implements device shadow mechanism, semantically identical to
to the Azure device twin and Amazon IoT device shadow. Google IoT Core
state/config objects provide similar functionality, but implemented quite differently.

Device shadow is a JSON object that is associated with a device. It
lives on a cloud, keeps device state, and always available
regardless whether the associated device is online or offline. Shadow object
has four top-level keys:

- `desired` - this is the state you want your device to have.
- `reported` - this is the state your device actually has.
- `delta` - automatically generated by the cloud
every time `desired` or `reported` changes.
`delta` is the difference between `desired` and `reported`
only for keys that are present in `desired`.
- `tags` - arbitrary settings invisible to the device but visible to
the cloud user.

The structure of the `desired` and `reported` subobjects is arbitrary -
create whatever structure you wish. However, the `dash` library that connects
your device to the dashboard, reserves some keys:

- `reported.ota` - object that keeps firmware information and last OTA status
- `reported.stats` - object that keeps device RAM statistics

The device shadow is displayed in the device list, and it is available for

edit in the device panel (when clicked on the device name):

Devices can update their shadow using C API or JavaScript API, see [shadow library](#) for reference.

In order to create new keys, send a shadow update with that new key and its value. In order to delete a key, send a shadow update where that key is set to `null`.

The best practice for using shadow on the device side is this:

- Catch the `MGOS_SHADOW_CONNECTED` event that indicates that the device is connected to its shadow, and report device current state to the `reported` key.
- Catch `MGOS_SHADOW_UPDATE_DELTA` event, and apply all keys in the `delta` to the device. If the state is changed, report and update to the `reported` key.

Example: see [example-shadow-js](#) app for the JavaScript example, and <https://github.com/cesanta/mongoose-os-smart-light> for C example.

Both implement the canonical best practice mentioned above.

OTA support



If a device includes `rpc-service-ota` library, then it can be updated remotely.

The three RPC functions that perform the OTA are `OTA.Begin`, `OTA.Write` and `OTA.End`.

mDash provides a convenience RESTful handler for the OTA, where you can just `POST` the new firmware .zip file, and mDash will call `OTA.Begin` followed by a sequence of `OTA.Write` calls, finished by `OTA.End`.

The OTA can be performed either via the Web UI, or programmatically using the REST API:

```
curl -H 'Authorization: Bearer API_KEY' \
-v -F file=@fw.zip
http://dash.mongoose-os.com/api/v2/devices/DEVICE_ID/ota
true
```

Once the firmware is updated, the device reboots in the "dirty", uncommitted state. An `OTA.Commit` call must be done to bless the new firmware, otherwise it will rollback, thinking that the health check did not pass. You can call `OTA.Commit` as any other RPC method. mDash provides a handy commit button for the convenience, when it sees an uncommitted device.

Data logging and visualisation

mDash provides an experimental support for data logging

How to enable data log on mDash

mDash can be instructed to save certain device shadow values into a persistant database. Currently, up to 100 data points can be saved. In the future, the number of datapoints will be unlimited.

In order to select which device shadow values get logged, modify the

`tags.save` array, which is the array of shadow keys to log.

By default, it is set to save only `ram_free` metric. That metric gets reported by the `demo-js` default firmware.

On mDash, click on the device name. In the device shadow editor, edit the `tags` value and save. For example, to log `ram_free` and `my_value`, do:

```
...
"tags": {
  "save": [
    "ram_free",
    "my_value"
  ],
},
...
```

How to log data from a device

Device logs data via the device shadow update. Here how it is done in C/C++:

```
# include "mgos.h"
<p class="mume-header" id="include-mgosh-5"></p>

# include "mgos_shadow.h" // make sure to include "shadow" lib in mos.yml
<p class="mume-header" id="include-mgos_shadowh-make-sure-to-include-shadow-lib-in-mosyml"></p>

// Somewhere in your code ...
mgos_shadow_updatef(0, "{my_value: %g}", 1.2345);
```

And in JavaScript:

```
load('api_shadow.js');

// Somewhere in your code ...
Shadow.update(0, JSON.stringify({my_value: 1.2345}));
```

See [Device shadow](#) section for the detailed device shadow explanation.

How to test data logging

You can easily simulate your device and log data from your workstation.

Essentially, logging data is updating a device shadow. You can update device shadow via the REST API by sending a `POST` request to the `/devices/:id` endpoint (don't forget to change both `API_KEY` and `DEVICE_ID`):

```
curl -H 'Content-Type: application/json' \
-H 'Authorization: Bearer API_KEY' \
-d '{"shadow": {"state": {"reported": {"ram_free": 12345}}}}' \
https://dash.mongoose-os.com/api/v2/devices/DEVICE_ID
```

- Mongoose OS

KIWI - a bluetooth beacon gateway

See <https://mdash.net/docs/quickstart/ble-gateway.md>

- [User guide](#)

Thank you for creating a pull request to contribute to Adafruit's GitHub code!

Before you open the request please review the following guidelines and tips to help it be more easily integrated:

- **Describe the scope of your change--i.e. what the change does and what parts of the code were modified.** This will help us understand any risks of integrating the code.
- **Describe any known limitations with your change.** For example if the change doesn't apply to a supported platform or the library please mention it.
- **Please run any tests or examples that can exercise your modified code.** We strive to not break users of the code and running tests/examples helps with this process.

Thank you again for contributing! We will try to test and integrate the change as soon as we can, but be aware we have many GitHub repositories to manage and can't immediately respond to every request. There is no need to bump or check in on a pull request (it will clutter the discussion of the request).

Also don't be worried if the request is closed or not integrated--sometimes the priorities of Adafruit's GitHub code (education, ease of use) might not match the priorities of the pull request. Don't fret, the open source community thrives on forks and GitHub makes it easy to keep your changes in a forked repo.

After reviewing the guidelines above you can delete this text from the pull request.

Thank you for opening an issue on an Adafruit Arduino library repository. To improve the speed of resolution please review the following guidelines and common troubleshooting steps below before creating the issue:

- **Do not use GitHub issues for troubleshooting projects and issues.** Instead use the forums at <http://forums.adafruit.com> to ask questions and troubleshoot why something isn't working as expected. In many cases the problem is a common issue that you will more quickly receive help from the forum community. GitHub issues are meant for known defects in the code. If you don't know if there is a defect in the code then start with troubleshooting on the forum first.
- **If following a tutorial or guide be sure you didn't miss a step.** Carefully check all of the steps and commands to run have been followed. Consult the forum if you're unsure or have questions about steps in a guide/tutorial.
- **For Arduino projects check these very common issues to ensure they don't apply:**
 - For uploading sketches or communicating with the board make sure you're using a **USB data cable** and **not a USB charge-only cable**. It is sometimes very hard to tell the difference between a data and charge cable! Try using the cable with other devices or swapping to another cable to confirm it is not the problem.
 - **Be sure you are supplying adequate power to the board.** Check the specs of

your board and plug in an external power supply. In many cases just plugging a board into your computer is not enough to power it and other peripherals.

- **Double check all soldering joints and connections.** Flakey connections cause many mysterious problems. See the [guide to excellent soldering](#) for examples of good solder joints.
- **Ensure you are using an official Arduino or Adafruit board.** We can't guarantee a clone board will have the same functionality and work as expected with this code and don't support them.

If you're sure this issue is a defect in the code and checked the steps above please fill in the following fields to provide enough troubleshooting information.
You may delete the guideline and text above to just leave the following details:

- Arduino board: **INSERT ARDUINO BOARD NAME/TYPE HERE**
- Arduino IDE version (found in Arduino -> About Arduino menu): **INSERT ARDUINO VERSION HERE**
- List the steps to reproduce the problem below (if possible attach a sketch or copy the sketch code in too): **LIST REPRO STEPS BELOW**

Cesanta note

Imported from https://github.com/adafruit/Adafruit_SSD1306/commit/ddfec78fa15f0ff8dfc8a76524077ba6bb5fc6f3

Adafruit_SSD1306

Compatibility

MCU	Tested Works	Doesn't Work	Not Tested	Notes
Atmega328 @ 16MHz	X			
Atmega328 @ 12MHz	X			
Atmega32u4 @ 16MHz	X			
Atmega32u4 @ 8MHz	X			
ESP8266	X			change OLED_RESET to different pin if using default I2C pins D4/D5.
Atmega2560 @ 16MHz	X			
ATSAM3X8E	X			
ATSAM21D	X			
ATtiny85 @ 16MHz		X		
ATtiny85 @ 8MHz		X		
Intel Curie @			X	

32MHz

STM32F2

X

- ATmega328 @ 16MHz : Arduino UNO, Adafruit Pro Trinket 5V, Adafruit Metro 328, Adafruit Metro Mini
- ATmega328 @ 12MHz : Adafruit Pro Trinket 3V
- ATmega32u4 @ 16MHz : Arduino Leonardo, Arduino Micro, Arduino Yun, Teensy 2.0
- ATmega32u4 @ 8MHz : Adafruit Flora, Bluefruit Micro
- ESP8266 : Adafruit Huzzah
- ATmega2560 @ 16MHz : Arduino Mega
- ATSAM3X8E : Arduino Due
- ATSAM21D : Arduino Zero, M0 Pro
- ATTiny85 @ 16MHz : Adafruit Trinket 5V
- ATTiny85 @ 8MHz : Adafruit Gemma, Arduino Gemma, Adafruit Trinket 3V

Arduino Adafruit SSD1306 library for Mongoose OS

This library provides support for
[1-wire](#) protocol for Mongoose OS.

Certificate Authority bundle

Overview

This library provides a single file to a device's filesystem: `fs/ca.pem`.
It contains CA certificates used by the most popular services, like
AWS, Google, Microsoft, etc.

RPC support for Azure

Direct Method RPC channel

Converts Azure [Direct Method](#) calls into mOS RPC calls.

Azure DM method name becomes RCP method name, payload becomes method args.

Since DM is a one way mechanism (cloud calls device), outgoing requests are rejected.

Cloud Messaging

It could be possible to support RPC over [cloud-to-device and device-to-cloud messaging](#).

Unlike DM, it could be bi-directional, with device initiating the request.

This is not implemented yet.

HTTP Server for Mongoose OS

Ethernet support

ESP32

ESP32 includes an Ethernet MAC and requires an external PHY, connected over RMII interface.

Two PHY models are currently supported:

- Microchip LAN87x0 ([LAN8710](#) supports both MII and RMII)
- [LAN8720](#) is RMII only) and [TI TLK110](#). PHY model selection is a compile-time option and is set [here](#).

There is a [detailed article](#)

on how to connect a PHY yourself. It is much easier to buy a dev board which already has one, for example,

[Olimex ESP32-EVB](#).

Once wired, `mos config-set eth.enable=true` to enable Ethernet (see below).

STM32

ESP32 includes an Ethernet MAC and requires an external PHY, connected over RMII interface.

Currently only LAN8742a PHY in RMII mode is supported.

MAC pin configuration

Since ETH MAC functions can be assigned to different pins, you need to specify pin assignments.

These are provided as C macros and can be specified in the `cdefs` section.

See example definitions for NUCLEO and Discovery boards in `mos_stm32.yml`.

Configuration

Common settings

```
"eth": {  
    "enable": true          // Enable Ethernet support  
}
```

ESP32 specific

```
"eth": {  
    "phy_addr": 0,          // RMII PHY address  
    "mdc_gpio": 23,         // GPIO to use for RMII MDC signal  
    "mdio_gpio": 18         // GPIO to use for RMII MDIO signal  
}
```

Note: the defaults match the EVB-ESP32 board, so if you use that, you won't need to adjust anything except setting `eth.enable` to `true`.

How to contribute

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

MCP23008/MCP23017 I2C and MCP23S08/MCP23S17 SPI Driver

A Mongoose library for MCP23XXX, a popular and cheap set of GPIO extenders using either I2C (MCP320XX) or SPI (MCP32SXX)

The MCP23008 is an 8-port device, and the MCP23017 is a 16-port device, but they are otherwise identical.

NOTE: SPI driver is TODO at this point. Contact the author for details.

Implementation details

The MCP230XX is a bi-directional GPIO with input pullup capability.

Each input pin can be explicitly set as INPUT or OUTPUT pin. For INPUT pins, the pin can be left floating or pulled up with a 100kOhm internal resistor.

Interrupts are handled with this driver. This allows for callbacks to be set for INPUT pin state changes.

NOTE: For simplicity reasons, on the MCP23x17 chips, intA and intB are combined, so it does not matter which of the pins are used on the chip, any port/pin state change will fire an interrupt, and only one pin needs to be used on the MCU to handle all 16 ports.

API Description

To start, `mgos_mcp23xxx_create()` is called with the correct I2C bus and address (by default 0x20), and optionally a GPIO pin on the microcontroller that serves as an interrupt pin, to detect MCP230XX input state changes.

NOTE: When the driver starts, it polls the current state from the chip without changing any ports. The benefit of this is that the MCU can safely reboot without loss of the GPIO state in MCP230XX.

The API follows `mgos_gpio.h` closely, enabling ports to be set as input or output.

For input ports, `mgos_mcp23xxx_gpio_set_pull()` can be called with either `MGOS_GPIO_PULL_NONE` or `MGOS_GPIO_PULL_UP`, but not with `MGOS_GPIO_PULL_DOWN` which returns an error, as the chip does not support input pulldowns.

Notably, `mgos_mcp23xxx_gpio_set_int_handler()` and `mgos_mcp23xxx_gpio_set_button_handler()` work identically to the `mgos_gpio_*` variants.

Example application

```

#include "mgos.h"
<p class="mume-header" id="include-mgosh-6"></p>

#include "mgos_config.h"
<p class="mume-header" id="include-mgos_configh"></p>

#include "mgos_mcp23xxx.h"
<p class="mume-header" id="include-mgos_mcp23xxxh"></p>

static void button_cb(int pin, void *user_data) {
    struct mgos_mcp23xxx *d = (struct mgos_mcp23xxx *)user_data;
    LOG(LL_INFO, ("GPIO[%d] callback, value=%d", pin, mgos_mcp23xxx_gpio_read(d, pin)));
    mgos_mcp23xxx_gpio_toggle(d, pin+8);
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_mcp23xxx *d;
    int i;

    if (!(d = mgos_mcp23017_create(mgos_i2c_get_global(), mgos_sys_config_get_mcp23xxx_i2caddr(),
                                    mgos_sys_config_get_mcp23xxx_int_gpio()))){
        LOG(LL_ERROR, ("Could not create MCP230XX"));
        return MGOS_APP_INIT_ERROR;
    }

    for(i=0; i<4; i++) mgos_mcp23xxx_gpio_setup_input(d, i, MGOS_GPIO_PULL_UP);
    for(i=8; i<16; i++) mgos_mcp23xxx_gpio_set_mode(d, i, MGOS_GPIO_MODE_OUTPUT);

    mgos_mcp23xxx_gpio_set_button_handler(d, 0, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_NEG, 10, button_cb, d);
    mgos_mcp23xxx_gpio_set_button_handler(d, 1, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_POS, 10, button_cb, d);
    mgos_mcp23xxx_gpio_set_button_handler(d, 2, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_ANY, 10, button_cb, d);
    mgos_mcp23xxx_gpio_set_button_handler(d, 3, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_ANY, 10, button_cb, d);

    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

RPC Service - I2C

This service provides an ability to manage I2C peripherals on devices remotely.
It is possible to call this service programmatically via serial, HTTP/RESTful,
WebSocket, MQTT or other transports
(see [RPC section](#)) or via the `mos` tool.

Below is a list of exported RPC methods and arguments:

I2C.Scan

Scan the I2C bus, return a list of available device addresses. Arguments: none.

Example usage:

```
mos call I2C.Scan
[
  31    # There is only 1 device with address 31 attached to the I2C bus
]
```

I2C.Read

Read data from the I2C device. Arguments:

```
{
  "addr": 31,      // Required. Device address.
  "len": 2         // Required. Number of bytes to read.
}
```

Reply:

```
{
  // Hex-encoded data. Each byte is encoded as XX hex code, e.g. 0x00 0x1d:
  "data_hex": "001d"
}
```

Example usage:

```
mos call I2C.Read '{"addr": 31, "len": 2}'
{
  "data_hex": "001d"
}
```

I2C.Write

Write data to the I2C device. Arguments:

```
{
  "addr": 31,          // Required. Device address.
  "data_hex": "1f3c6a" // Required. Hex-encoded data to write
}
```

Example usage (showing failed write):

```
mos call I2C.Write '{"addr": 31, "data_hex": "1f3c6a"}'
Error: remote error: I2C write failed
```

I2C.ReadRegB

Read 1-byte register value. Arguments:

```
{
  "addr": 31,  // Required. Device address.
  "reg": 0     // Required. Register number.
}
```

Example usage:

```
mos call I2C.ReadRegB '{"addr": 31, "reg": 0}'
{
  "value": 0
}
```

I2C.WriteRegB

Write 1-byte register value. Arguments:

```
{  
  "addr": 31,      // Required. Device address.  
  "reg": 0,        // Required. Register number.  
  "value": 0       // Required. 1-byte value to write.  
}
```

Example usage:

```
mos call I2C.WriteRegB '{"addr": 31, "reg": 0, "value": 0}'
```

I2C.ReadRegW

Same as `I2C.ReadRegB`, but read 2-byte (word) register value.

I2C.WriteRegW

Same as `I2C.WriteRegB`, but write 2-byte (word) register value.

RPC ATCA Service support

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or

feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.

2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

Original driver

This code is derived from [Sparkfun's driver](#) which was placed in the public domain. Please consider supporting Sparkfun Electronics by picking up a few boards from them!

APDS-960 I2C Driver

A Mongoose library for Avago's APDS-9960 RGB, IR and Gesture Sensor.

Implementation details

The APDS-9960 device features advanced Gesture detection, Proximity detection, Digital Ambient Light Sense (ALS) and Color Sense (RGBC). The slim modular package, L 3.94 x W 2.36 x H 1.35 mm, incorporates an IR LED and factory calibrated LED driver for drop-in compatibility with existing footprints.

Gesture detection

Gesture detection utilizes four directional photodiodes to sense reflected IR energy (sourced by the integrated LED) to convert physical motion information (i.e. velocity, direction and distance) to a digital information. The architecture of the gesture engine features automatic activation (based on Proximity engine results), ambient light subtraction, cross-talk cancelation, dual 8-bit data converters, power saving inter-conversion delay, 32-dataset FIFO, and interrupt driven I2C communication. The gesture engine accommodates a wide range of mobile device gesturing requirements: simple UP-DOWN-RIGHT-LEFT gestures or more complex gestures can be accurately sensed. Power consumption and noise are minimized with adjustable IR LED timing.

Proximity detection

The Proximity detection feature provides distance measurement (E.g. mobile device screen to user's ear) by photodiode detection of reflected IR energy (sourced by the integrated LED). Detect/release events are interrupt driven, and occur whenever proximity result crosses upper and/ or lower threshold settings. The proximity engine features offset adjustment registers to compensate for system offset caused by unwanted IR energy reflections appearing at the sensor. The IR LED intensity is factory trimmed to eliminate the need for end-equipment calibration due to component variations. Proximity results are further improved by automatic ambient light subtraction.

Color and ALS detection

The Color and ALS detection feature provides red, green, blue and clear light intensity data. Each of the R, G, B, C channels have a UV and IR blocking filter and a dedicated data converter producing 16-bit data simultaneously. This architecture allows applications to accurately measure ambient light and sense color which enables devices to calculate color temperature and control display backlight.

API Description

There are two APIs defined in this driver. Firstly, a low level API is used to drive the configuration of the chip. Secondly, a higher level abstraction is provided for application programmers that want to merely read proximity and light data from the sensor, or install a hardware interrupt to callback user code upon certain events (such as proximity, light and gesture events described above.)

Notes

Gesture sending is incredibly hard with this sensor. Work is needed to improve the performance of the gesture code in this driver.

Proximity and Light sensing and interrupts are working fine.

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```

#include "mgos.h"
<p class="mume-header" id="include-mgosh-7"></p>

#include "mgos_config.h"
<p class="mume-header" id="include-mgos_configh-1"></p>

#include "mgos_apds9960.h"
<p class="mume-header" id="include-mgos_apds9960h"></p>

static void light_interrupt(uint16_t clear, uint16_t red, uint16_t green, uint16_t blue) {
    LOG(LL_INFO, ("clear=%u red=%u green=%u blue=%u", clear, red, green, blue));
}

static void proximity_interrupt(uint8_t proximity) {
    LOG(LL_INFO, ("proximity=%u", proximity));
}

static void apds9960_timer_cb(void *user_data) {
    struct mgos_apds9960 *sensor = (struct mgos_apds9960 *)user_data;
    uint16_t clear = 0, red = 0, green = 0, blue = 0;
    uint8_t proximity = 0;

    if (!mgos_apds9960_read_light(sensor, &clear, &red, &green, &blue)) {
        LOG(LL_ERROR, ("Could not read APDS9960 light sensor"));
    }
    if (!mgos_apds9960_read_proximity(sensor, &proximity)) {
        LOG(LL_ERROR, ("Could not read APDS9960 proximity sensor"));
    }

    LOG(LL_INFO, ("clear=%u red=%u green=%u blue=%u proximity=%u", clear, red, green, blue, proximity));
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_apds9960 *sensor;

    sensor = mgos_apds9960_create(mgos_i2c_get_global(), mgos_sys_config_get_apds9960_i2caddr());
    if (!sensor) {
        LOG(LL_ERROR, ("Could not create APDS9960 sensor"));
        return false;
    }

    if (!mgos_apds9960_set_callback_light(sensor, 10, 1000, light_interrupt)) {
        LOG(LL_ERROR, ("Could not enable APDS9960 light callback"));
        return false;
    }

    if (!mgos_apds9960_set_callback_proximity(sensor, 0, 150, proximity_interrupt)) {
        LOG(LL_ERROR, ("Could not enable APDS9960 proximity callback"));
        return false;
    }

    mgos_set_timer(1000, true, apds9960_timer_cb, sensor);

    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

JS bindings for ESP32 touch pad sensor

Overview

This library provides JavaScript bindings for the ESP32 touch pad sensor.

The JS API largely mirrors the [C API](#).

Examples

Polling the sensor manually

```
load('api_esp32_touchpad.js');

// Touch sensors are numbered from 0 to 9.
// For convenience, TouchPad.GPIO map translates from GPIO number to sensor number.
let ts = TouchPad.GPIO[15];

TouchPad.init();
TouchPad.setVoltage(TouchPad.HVOLT_2V4, TouchPad.LVOLT_0V8, TouchPad.HVOLT_ATTEN_1V5);
TouchPad.config(ts, 0);
Timer.set(1000 /* 1 sec */, Timer.REPEAT, function() {
  let tv = TouchPad.read(ts);
  print('Sensor', ts, 'value', tv);
}, null);
```

Using interrupts

```
load('api_esp32_touchpad.js');

// Touch sensors are numbered from 0 to 9.
// For convenience, TouchPad.GPIO map translates from GPIO number to sensor number.
let ts = TouchPad.GPIO[15];

TouchPad.init();
TouchPad.filterStart(10);
TouchPad.setMeasTime(0x1000, 0xffff);
TouchPad.setVoltage(TouchPad.HVOLT_2V4, TouchPad.LVOLT_0V8, TouchPad.HVOLT_ATTEN_1V5);
TouchPad.config(ts, 0);
Sys.usleep(100000); // wait a bit for initial filtering.
let noTouchVal = TouchPad.readFiltered(ts);
let touchThresh = noTouchVal * 2 / 3;
print('Sensor', ts, 'noTouchVal', noTouchVal, 'touchThresh', touchThresh);
TouchPad.setThresh(ts, touchThresh);
TouchPad_isrRegister(function(st) {
  // st is a bitmap with 1 bit per sensor.
  let val = TouchPad.readFiltered(ts);
  print('Status:', st, 'Value:', val);
}, null);
TouchPad.intrEnable();
```

Mongoose OS Fingerprint Library

This library provides a simple API that describes a popular set of serial (UART) fingerprint modules made by Grow.

Application Interface

The fingerprint module consists of an ARM microcontroller and an optical or capacitive sensor unit, and it uses a Serial UART to communicate, by default at 57600 baud, with its host.

There are two modes of operation, command and data:

1. The command protocol is a serial transaction where the host writes a packet with a command, and the unit returns a packet with a response.
2. The data protocol is a repeated set of data packets, followed by an end-of-data packet.

There are two main functions that each fingerprint module exposes: enrolling fingerprints and matching fingerprints.

Enrolling fingerprints

First, two separate images are taken of the fingerprint, each stored in a memory bank. Then, they are combined into a *model* (also called a *template*), if the images are sufficiently similar. If these operations are successful, the resulting *model* can be stored into flash using a model slot.

Matching fingerprints

In this mode, an image is taken of the fingerprint, and the internal flash database is searched for a match. If a match is found, the resulting fingerprint id and quality score are returned. If no match was found (either because the fingerprint was not cleanly imaged, or if it simply does not exist in the database), an error condition is returned.

Lowlevel API primitives

First, a device is created with `mgos_fingerprint_create()`, passing in a `struct mgos_fingerprint_cfg` configuration, which can be pre-loaded with defaults by calling `mgos_fingerprint_config_set_defaults()`, after which an opaque handle is returned to the driver object. When the driver is no longer needed, `mgos_fingerprint_destroy()` can be called with a pointer to the handle, after which it will be cleaned up and memory returned.

Each API call returns a signed integer, signalling `MGOS_FINGERPRINT_OK` upon success and some other value (see `include/mgos_fingerprint.h`), negative numbers for errors, and positive numbers for successful API calls with negative results (like no finger image taken, no fingerprint found in database, etc).

Then, the following primitives are implemented:

- `mgos_fingerprint_image_get()`: This fetches an image of a fingerprint on the sensor, and stores it in RAM for further processing.
- `mgos_fingerprint_image_genchar()`: This turns the image of the fingerprint into a character array and stores it in one of two *slots*.
- `mgos_fingerprint_model_combine()`: This compares two character arrays (which represent fingerprint images), and combines them into a *model*. If the fingerprints were sufficiently similar, the operation is successful, otherwise an error is returned.
- `mgos_fingerprint_model_store()`: This stores the combined model into a position of the flash memory of the module.
- `mgos_fingerprint_database_erase()`: This erases all models from flash memory.
- `mgos_fingerprint_database_search()`: This compares an image of a fingerprint with the saved models in flash memory, returning the model number and a quality score upon success, and an error otherwise.

Some models (R502, R503, for example) may also support lighting operations:

- `mgos_fingerprint_led_on()`: This turns the LED in the sensor device on.
- `mgos_fingerprint_led_off()`: This turns the LED in the sensor device off.

- `mgos_fingerprint_led_aura()`: This uses a ring-LED around the sensor device in one of multiple colors (typically red, blue or purple) and either flashing N times, fading in or out, or swelling N times).

Library primitives

In addition to the low level primitives that the API provides, there is also a higher level implementation available. By calling `mgos_fingerprint_svc_init()`, a timer is set up at `period_ms` milliseconds intervals, at which time a fingerprint image is attempted to be created. Based on the *mode* of operation (which can be set by `mgos_fingerprint_svc_mode_set()` to either *match* or *enroll*), the fingerprint is processed accordingly.

A callback handler in `struct mgos_fingerprint_cfg` receives event callbacks as follows:

- `MGOS_FINGERPRINT_EV_INITIALIZED`: when the chip is first initialized successfully.
- `MGOS_FINGERPRINT_EV_IMAGE`: each time the sensor has successfully fetched an image.
- `MGOS_FINGERPRINT_EV_MATCH_OK`: in *match mode* each time an image matched with one of the model entries in the flash database. The matched fingerprint ID and score are packed into `*ev_data`, the top 16 bits are the `score`, the lower 16 bits are the `finger_id`.
- `MGOS_FINGERPRINT_EV_MATCH_ERROR`: in *match mode* each time an image did not match with any model entries in the flash database, or if a processing error occurred.
- `MGOS_FINGERPRINT_EV_STATE_MATCH`: when *match mode* is entered.
- `MGOS_FINGERPRINT_EV_STATE_ENROLL1`: when *enroll mode* is processing the first (of two) fingerprint images.
- `MGOS_FINGERPRINT_EV_STATE_ENROLL2`: when *enroll mode* is processing the second (of two) fingerprint images.
- `MGOS_FINGERPRINT_EV_ENROLL_OK`: when *enroll mode* successfully stored a fingerprint model in the flash database. The stored fingerprint ID is packed into `*ev_data`, the lower 16 bits are the `finger_id`.
- `MGOS_FINGERPRINT_EV_ENROLL_ERROR`: when *enroll mode* failed to process or store a fingerprint model.

Supported devices

Popular GROW devices are supported, look for Grow sensors [on Aliexpress](#).

Most of them are named **Rxxx** with a three-digit number, like **R503**,

R300 or **R307**. There is also a unit for sale at [Adafruit](#).

Example Code

For a complete demonstration of the driver, look at this [Mongoose App](#).

HD44780 library for Mongoose OS

A library used to control a HD44780 LCD display in a MongooseOS project, without an I2C module that is commonly attached to the LCD. It should use 6 GPIO pins in total, that should be mapped in config_schema like this example:

```
- ["HD44780.GPIO.D7" , 14 ]
- ["HD44780.GPIO.D6" , 12 ]
- ["HD44780.GPIO.D5" , 13 ]
- ["HD44780.GPIO.D4" , 15 ]
- ["HD44780.GPIO.E" , 4 ]
- ["HD44780.GPIO.RS" , 2 ]
- ["HD44780.PHYSICAL_NUMBER_OF_LINES" , 4 ]
```

Also, `HD44780.PHYSICAL_NUMBER_OF_LINES` is expected to be 2(default) or 4. I don't think there is models with a different number of lines.

About the other LCD pins:

- pins D0,D1,D2 and D3 are not necessary since this library uses the 4-bit interface mode.
- pin R/W should be low (GND) all the time, since this library doesn't support reading commands from display

Configuration

```
config_schema:
- ["HD44780" , "o" , {"title": "HD44780 configuration schema"}]
- ["HD44780.PHYSICAL_NUMBER_OF_LINES" , "i" , 2 , {"title": "HD44780 model available number of lines. Expected 2 or 4."}]
- ["HD44780.GPIO" , "o" , {"title": "ESP8266 GPIO pins used in HD44780's pins"}]
- ["HD44780.GPIO.D7" , "i" , 14 , {"title": "ESP8266's GPIO pin used in HD44780's D7 pin"}]
- ["HD44780.GPIO.D6" , "i" , 12 , {"title": "ESP8266's GPIO pin used in HD44780's D6 pin"}]
- ["HD44780.GPIO.D5" , "i" , 13 , {"title": "ESP8266's GPIO pin used in HD44780's D5 pin"}]
- ["HD44780.GPIO.D4" , "i" , 15 , {"title": "ESP8266's GPIO pin used in HD44780's D4 pin"}]
- ["HD44780.GPIO.E" , "i" , 4 , {"title": "ESP8266's GPIO pin used in HD44780's E pin"}]
- ["HD44780.GPIO.RS" , "i" , 2 , {"title": "ESP8266's GPIO pin used in HD44780's RS pin"}]
```

Example

This is a simple C program example using a 4-line model.

Don't forget to set `HD44780.PHYSICAL_NUMBER_OF_LINES` config equal to 4 in this case.

main.c

```

# include "mgos.h"
<p class="mume-header" id="include-mgosh-8"></p>

# include "mgos_hd44780.h"
<p class="mume-header" id="include-mgos_hd44780h"></p>

enum mgos_app_init_result mgos_app_init(void) {

    lcd_write("First line!");

    lcd_move_cursor(19,1); // Moves to the final of the second line
    lcd_entry_mode_set( LCD_DIRECTION_DECREMENT , LCD_DISPLAY_SHIFT_OFF );
    lcd_write("Second line?"); // Writes "Second line?" backwards because entry mode decrementing cursor

    lcd_move_cursor(0,2); // Moves to the beggining of the third line
    lcd_entry_mode_set( LCD_DIRECTION_INCREMENT , LCD_DISPLAY_SHIFT_OFF );

    // Writes 1 2 3, spaced because cursor shifting
    lcd_write("1");
    lcd_shift_cursor(LCD_SHIFT_RIGHT);
    lcd_write("2");
    lcd_shift_cursor(LCD_SHIFT_RIGHT);
    lcd_write("3");

    lcd_move_cursor(9,3); // Moves to the middle of the fourth line
    lcd_write("Bye...");

    lcd_move_cursor(0,3);

    return MGOS_APP_INIT_SUCCESS;
}

```

mos.yml

```

config_schema:
  - ["HD44780.PHYSICAL_NUMBER_OF_LINES" , 4]

libs:
  - origin: https://github.com/rilpires/mgos_hd44780_controller
    name: hd44780

manifest_version: 2017-05-18

```

Notes

Don't forget to add:

```
  name: hd44780
```

in libs section.

There is some functionalities yet to implement:

- Set the number of lines used to half of the available lines. I don't think this is useful at all but it is necessary in order to use the 5x10 font (it is basically the same font with an underline below it)
- Implement reading commands from the display. It would be necessary an additional GPIO pin.
- Implement creating custom characters.

How to contribute

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

ADS1X1X I2C Driver

A Mongoose library for various I2C speaking ADCs from Texas Instruments:

- ADS1115 - 16bit, 860 Samples/sec, 2 differential / 4 single-ended, programmable gain
- ADS1114 - 16bit, 860 Samples/sec, 1 differential / 1 single-ended, programmable gain
- ADS1113 - 16bit, 860 Samples/sec, 1 differential / 1 single-ended, no gain
- ADS1015 - 12bit, 3300 Samples/sec, 2 differential / 4 single-ended, programmable gain
- ADS1014 - 12bit, 3300 Samples/sec, 1 differential / 1 single-ended, programmable gain

- ADS1013 - 12bit, 3300 Samples/sec, 1 differential / 1 single-ended, no gain.

The most common are the **ADS1115** and **ADS1015** chips.

Implementation details

The driver takes care of exposing the correct functionality based on which **type** is created. Differential measurements can be taken on all devices, but only **ADS1x15** has multiple options.

API Description

First, create a device using `mgos_ads1x1x_create()` by specifying the type of chip you're using. Take some measurements using `mgos_ads1x1x_read()`, and clean up the driver by using `mgos_ads1x1x_destroy()`.

`mgos_ads1x1x_set_fsr()` is used to set the full scale range (FSR) of the ADC. Each chip supports ranges from 6.144 Volts down to 0.256 Volts. You can read the current FSR with `mgos_ads1x1x_get_fsr()`.

`mgos_ads1x1x_set_dr()` is used to set the data rate of continuous measurements. The support differs between **ADS101X** (the 12-bit version, which is faster), and **ADS111X** (the 16-bit version, which is slower). You can read the current DR with `mgos_ads1x1x_get_dr()`.

`mgos_ads1x1x_read()` starts a singleshot measurement on the given channel (which takes 1ms for **ADS101X** and 8ms for **ADS111X**), and returns a 16 bit signed value. The datasheet mentions that with input voltages around **GND**, a negative value might be returned (ie -2 rather than 0).

`mgos_ads1x1x_read_diff()` starts a singleshot measurement of the differential voltage between two channels, typically **Chan0** and **Chan1**. Several channel pairs are allowed, see the include file for details. Note, that this function is only available on **ADS1X15** chips.

Example application

```

# include "mgos.h"
<p class="mume-header" id="include-mgosh-9"></p>

# include "mgos_config.h"
<p class="mume-header" id="include-mgos_configh-2"></p>

# include "mgos_ads1x1x.h"
<p class="mume-header" id="include-mgos_ads1x1xh"></p>

void timer_cb(void *data) {
    struct mgos_ads1x1x *d = (struct mgos_ads1x1x *)data;
    int16_t res[4];

    if (!d) return;

    for(int i=0; i<4; i++) {
        if (!mgos_ads1x1x_read(s_adc, i, &res[i])) {
            LOG(LL_ERROR, ("Could not read device"));
            return;
        }
    }
    LOG(LL_INFO, ("chan=%{#d, #d, #d, #d}", res[0], res[1], res[2], res[3]));
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_ads1x1x *d = NULL;

    if (!(d = mgos_ads1x1x_create(mgos_i2c_get_global(), 0x48, ADC_AMG3305))) {
        LOG(LL_ERROR, ("Could not create AMG3305"));
        return MGOS_APP_INIT_ERROR;
    }

    mgos_set_timer(100, true, timer_cb, d);

    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Use mDash API to set a device label and create the UI

Example 1

```

# include "mgos.h"
<p class="mume-header" id="include-mgosh-10"></p>

# define USE_WIFI 0
<p class="mume-header" id="define-use_wifi-0"></p>

# if USE_WIFI == 1
<p class="mume-header" id="if-use_wifi-1"></p>

# include "mgos_wifi.h"
<p class="mume-header" id="include-mgos_wifih"></p>

```

```

# endif
<p class="mume-header " id="endif"></p>

# include "mgos_mdash_api.h"
<p class="mume-header " id="include-mgos_mdash_apih"></p>

static void create_mdash_ui() {
    struct mgos_mdash_widgets *widgets = mgos_mdash_widgets_create(0);
    mgos_mdash_widgets_add_widget(
        widgets, mgos_mdash_widget_toggle_create("Switch LED on/off",
                                                "state.reported.on"));
    mgos_mdash_widgets_add_widget(
        widgets,
        mgos_mdash_widget_value_create("Uptime (s):", "state.reported.uptime"));
    mgos_mdash_widgets_add_widget(
        widgets, mgos_mdash_widget_input_create(
            "Led pin", "state.reported.led_pin", "fa-save"));
    mgos_mdash_widgets_add_widget(
        widgets, mgos_mdash_widget_button_create("Reboot", "Sys.Reboot", NULL,
                                                "fa-power-off"));

    mgos_mdash_create_ui(widgets);
    mgos_mdash_widgets_free(widgets);
}

# if USE_WIFI == 1
<p class="mume-header " id="if-use_wifi-1-1"></p>

static void wifi_cb(int ev, void *evd, void *arg) {
    switch (ev) {
        case MGOS_WIFI_EV_STA_IP_ACQUIRED: {
            LOG(LL_INFO, ("%s", "MGOS_WIFI_EV_STA_IP_ACQUIRED"));

            mgos_mdash_set_label("my_label");
            create_mdash_ui();

            break;
        }
    }
    (void) evd;
    (void) arg;
}
# else
<p class="mume-header " id="else"></p>

static void net_cb(int ev, void *evd, void *arg) {
    switch (ev) {
        case MGOS_NET_EV_IP_ACQUIRED: {
            LOG(LL_INFO, ("%s", "MGOS_NET_EV_IP_ACQUIRED"));

            mgos_mdash_set_label("my_label");
            create_mdash_ui();
            break;
        }
    }
    (void) evd;
    (void) arg;
}
# endif
<p class="mume-header " id="endif-1"></p>

enum mgos_app_init_result mgos_app_init(void) {
# if USE_WIFI == 1
<p class="mume-header " id="if-use_wifi-1-2"></p>

    mgos_event_add_group_handler(MGOS_EVENT_GRP_WIFI, wifi_cb, NULL);
# else
<p class="mume-header " id="else-1"></p>

```

```

mgos_event_add_group_handler(MGOS_EVENT_GRP_NET, net_cb, NULL);
# endif
<p class="mume-header" id="endif-2"></p>

return MGOS_APP_INIT_SUCCESS;
}

```

Example 2

- Define the widgets in the `mos.yml` of your application, e.g.

```

config_schema:
# mDash widgets
<p class="mume-header" id="mdash-widgets"></p>

- ["mdash.toggle.title", "Switch LED on/off"]
- ["mdash.toggle.key", "state.reported.on"]

- ["mdash.value.title", "Uptime (s):"]
- ["mdash.value.key", "state.reported.uptime"]

- ["mdash.value1.enable", true]    # enable this widget
- ["mdash.value1.title", "Min free RAM:"]
- ["mdash.value1.key", "state.reported.ram_min_free"]

- ["mdash.value2.enable", true]    # enable this widget
- ["mdash.value2.title", "Free RAM:"]
- ["mdash.value2.key", "state.reported.ram_free"]

- ["mdash.value3.enable", true]    # enable this widget
- ["mdash.value3.title", "Temp:"]
- ["mdash.value3.key", "state.reported.temp"]

- ["mdash.button.title", "Reboot"]
- ["mdash.button.method", "Sys.Reboot"]
- ["mdash.button.params", ""]
- ["mdash.button.icon", "fa-power-off"]

```

- The code

```

# include "mgos.h"
<p class="mume-header" id="include-mgosh-11"></p>

# include "mgos_mdash_api.h"
<p class="mume-header" id="include-mgos_mdash_apih-1"></p>

static void create_mdash_ui() {
    struct mgos_mdash_widgets *widgets;
    if (mgos_mdash_create_widgets_from_config(&widgets)) {
        mgos_mdash_create_ui(widgets);
    }
    mgos_mdash_widgets_free(widgets);
}

static void cloud_cb(int ev, void *evd, void *arg) {
    switch (ev) {
        case MGOS_EVENT_CLOUD_CONNECTED: {
            LOG(LL_INFO, ("%s - Cloud connected", __FUNCTION__));
            mgos_mdash_set_label("my_label");
            create_mdash_ui();
            break;
        }
    }
    (void) evd;
    (void) arg;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_event_add_handler(MGOS_EVENT_CLOUD_CONNECTED, cloud_cb, NULL);

    return MGOS_APP_INIT_SUCCESS;
}

```

- Drawback - the order of creating the widgets is fixed: value, input, toggle and button.
- Advantage - a widget can be disabled at runtime. E.g.

```
mos --port <serial,ws,mqtt port> config-set mdash.value3.enable=false
```

Usage

- Create a device in mDash
- Create a API key
- Build and flash the application
- mos wifi SSID PASSWORD
- Provision the device in mDash
- mos --port <port> config-set mdash.device_id=<device ID from mDash> mdash.api_key=<API key from mDash>

Refs

[Device UI for remote control](#)

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

Software License Agreement (BSD License)

Copyright (c) 2012 Adafruit Industries. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE

IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Attribution

These are files taken from Adafruit's GFX library:

<https://github.com/adafruit/Adafruit-GFX-Library>

Modifications

- `gfxfont.h`: added a few attributes to `struct GFXfont`
- `fonts/*.h`: added the attributes from `struct GFXfont`
- `fonts/*.h`: wrapped the data in `#ifndef X; #define X; #endif` stanzas
- `src/mgos_il9341_primitives.c`: Rewrote from C++ class to standard C implementation. Also, ensured each function takes (x,y) arguments that are unsigned ints, as this implementation sets a window to draw in.

Attribution

This is a copy of uPNG taken from <https://github.com/elanthis/upng>

Modifications

None.

ILI9341 TFT Driver

Introduction

This library is a native implementation of a graphics chip for small TFT screens, the ILI9341. The chip is found in many popular displays, including 320x240 and 480x320 pixel versions, and is capable of driving 16-bit (RGB-565) as well as 18-bit (RGB-666) modes. Although the chip is capable of using 18-bit colors (262,114), that requires three bytes per pixel to be written. Therefore, this driver initializes it in 16-bit colors (65,536) allowing for two bytes per pixel.

Fundamentals

The chip allows access to its framebuffer memory by setting a window by means of (x0,y0)-(x1,y1) coordinates, and then writing pixel data. To implement higher level shapes and fonts, we implement three basic functions on the hardware:

```
void mgos_ilis341_drawPixel(uint16_t x0, uint16_t y0);
```

This function sets the window to be exactly 1 pixel and writes the current foreground color to it.

```
void mgos_ilis341_drawLine(uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1);
```

This function can draw a vertical line by setting the window to (x0,y0)-(x0+1,y1), and writing the current foreground color to it. It can draw a horizontal line similarly by setting the window to (x0,y0)-(x1,y0+1). Any other lines can be decomposed into segments of horizontal and vertical lines and single pixels.

```
void mgos_ilis341_fillRect(uint16_t x0, uint16_t y0, uint16_t w, uint16_t h);
```

This sets the window to (x0,y0)-(x0+w-1,y0+h-1) and writes the current foreground color to it.

Primitives

Orientations

```
void mgos_ilis341_set_orientation(uint8_t madctl, uint16_t rows, uint16_t cols);
```

Depending on how the hardware manufacturer connected the LCD panel to the ILI9341 chip, several registers are provided to determine the true orientation of the screen. From the datasheet, there are 5 bits which determine that orientation, in the `MADCTL` register, as follows:

- bit2 `ILI9341_MADCTL_MH` - sets the *Horizontal Refresh*, 0=Left-Right and 1=Right-Left
- bit4 `ILI9341_MADCTL_ML` - sets the *Vertical Refresh*, 0=Top-Bottom and 1=Bottom-Top
- bit5 `ILI9341_MADCTL_MV` - sets the *Row/Column Swap*, 0=Normal and 1=Swapped
- bit6 `ILI9341_MADCTL_MX` - sets the *Column Order*, 0=Left-Right and 1=Right-Left
- bit7 `ILI9341_MADCTL_MY` - sets the *Row Order*, 0=Top-Bottom and 1=Bottom-Top

These bits are defined as `ILI9341_MADCTL_*` in `mgos_ilis341.h` header file.

By means of example, here's a definition for Adafruit panels:

```
# define ADAFRUIT_PORTRAIT      (ILI9341_MADCTL_MX)
<p class="mume-header" id="define-adafruit_portrait-ili9341_madctl_mx"></p>

# define ADAFRUIT_LANDSCAPE     (ILI9341_MADCTL_MX|ILI9341_MADCTL_MY|ILI9341_MADCTL_MV)
<p class="mume-header" id="define-adafruit_landscape-ili9341_madctl_mx|ili9341_madctl_my|ili9341_madctl_mv"></p>

# define ADAFRUIT_PORTRAIT_FLIP (ILI9341_MADCTL_MY)
<p class="mume-header" id="define-adafruit_portrait_flip-ili9341_madctl_my"></p>

# define ADAFRUIT_LANDSCAPE_FLIP (ILI9341_MADCTL_MV)
<p class="mume-header" id="define-adafruit_landscape_flip-ili9341_madctl_mv"></p>
```

And here's a definition for the panel used in M5Stack:

```
# define M5STACK_PORTRAIT      (ILI9341_MADCTL_MV|ILI9341_MADCTL_MY)
<p class="mume-header " id="define-m5stack_portrait-ilil9341_madctl_mvili9341_madctl_my"></p>

# define M5STACK_LANDSCAPE     (0x00)
<p class="mume-header " id="define-m5stack_landscape-0x00"></p>

# define M5STACK_PORTRAIT_FLIP (ILI9341_MADCTL_MV|ILI9341_MADCTL_MX)
<p class="mume-header " id="define-m5stack_portrait_flip-ilil9341_madctl_mvili9341_madctl_mx"></p>

# define M5STACK_LANDSCAPE_FLIP (ILI9341_MADCTL_MY|ILI9341_MADCTL_ML|ILI9341_MADCTL_MX)
<p class="mume-header " id="define-m5stack_landscape_flip-ilil9341_madctl_myili9341_madctl_mlili9341_madctl_mx"></p>
```

The API call `mgos_ilil9341_set_orientation()` gives full control over these settings, as well as setting the resulting width and height in pixels.

Window and Clipping

The driver works by setting a bounding box around the area to be drawn in:

```
void mgos_ilil9341_set_window(uint16_t x0, uint16_t y0,
                               uint16_t x1, uint16_t y1);
```

Any operation will be homed within the window, and clipped if it runs outside of the window. For example, setting the window to (20,30)-(99,99) will create a bounding box of 80 pixels wide and 70 pixels high. Subsequently calling `mgos_ilil9341_drawPixel(0,0)` will draw a pixel at the top left of the window, which in this case is physical pixel (20,30). Drawing a pixel at coordinates (80,70) will draw a pixel at the bottom right corner of the window, whereas drawing one at (81,71) will not show anything.

Geometric Shapes

A set of primitives are provided to allow drawing of geometric shapes such as circles, triangles and boxes:

```
void mgos_ilil9341_drawRect(uint16_t x0, uint16_t y0, uint16_t w, uint16_t h);
void mgos_ilil9341_drawRoundRect(uint16_t x0, uint16_t y0, uint16_t w,
                                 uint16_t h, uint16_t r);
void mgos_ilil9341_fillRoundRect(uint16_t x0, uint16_t y0, uint16_t w,
                                 uint16_t h, uint16_t r);

void mgos_ilil9341_drawCircle(uint16_t x0, uint16_t y0, uint16_t r);
void mgos_ilil9341_fillCircle(uint16_t x0, uint16_t y0, uint16_t r);

void mgos_ilil9341_drawTriangle(uint16_t x0, uint16_t y0, uint16_t x1,
                                uint16_t y1, uint16_t x2, uint16_t y2);
void mgos_ilil9341_fillTriangle(uint16_t x0, uint16_t y0, uint16_t x1,
                                uint16_t y1, uint16_t x2, uint16_t y2);
```

Fonts

Fonts can be embedded in the program by including the font files in `fonts/*.h`. The width and height of a given string can be calculated, and a string can be printed (using the set foreground and background colors). As with geometric shapes, the print function is relative to the window, (0,0) being top-left, and will be clipped at the window borders.

Images

Since the ILI9341 chip expects pixels in 16-bit network byte order, we can

transmit a line of pixels from an image. Because the build target of the driver is all Mongoose available platforms (notably ESP8266, ESP32 and CC32xx), memory is an issue, and decompressing **PNG**, **GIF**, or **JPG** files can be CPU as well as memory intensive, a simplistic alternative is provided: **DIF**

This function can be used for displaying **DIF** images:

```
void mgos ili9341_drawDIF(uint16_t x0, uint16_t y0, char *fn);
```

DIF file format

DIF stands for **dumb image format**, and consists of a 16 byte header followed by raw pixel data, 16 bits per pixel, in network byte order. This alleviates CPU and memory pressure on the micro controller, at the expense of file size. The header is as follows:

- The first four bytes (offset 0..3) are the string **DIF\001**
- The second four bytes (offset 4..7) are the width (an **uint32_t** in network byte order).
- The third four bytes (offset 8..11) are the height (an **uint32_t** in network byte order).
- The byte at offset 12 denotes the raw image format that follows:
 - Value of 0 means: 16 bit **RGB-565** in network byte order.
- The last three bytes (offset 13..15) are reserved for future use.

The file size will therefore be **w * h + 16** bytes.

Converting PNG to DIF

To convert **PNG** images to **DIF** images, there is a **png2dif** utility available in this repo, under **contrib/png2dif**

Clone this repo, and while inside the **contrib/png2dif** directory, run **make** to generate **png2dif** executable. You can then run **png2dif -i input.png -o output.dif** to convert a non-transparent **png** file, to a **dif** file.

The generated **dif** file should then be placed inside the **fs** directory, and can be displayed using **mgos ili9341_drawDIF**. As an example, if the filename is **output.dif**, to display at x cord of 0 and y cord of 0:

```
mgos ili9341_drawDIF( 0, 0, "/output.dif" );
```

Example usage of **DIF** images, and this library, can be found in the [Huzzah Featherwing Example App](#)

Example Application

mos.yml

The driver uses the Mongoose native SPI driver. It is configured by setting up the **MOSI**, **MISO**, **SCLK** pins and assigning one of the three available **CS** positions, in this example (which was taken from the Huzzah32 ESP32 microcontroller), we are going to use **CS0**:

```
config_schema:  
- ["spi.enable", true]  
- ["spi.cs0_gpio", 15] # The ILI9341 CS pin  
- ["spi.cs1_gpio", -1]  
- ["spi.cs2_gpio", -1]  
- ["spi.mosi_gpio", 18]  
- ["spi.miso_gpio", 19]  
- ["spi.sclk_gpio", 5]  
- ["ili9341.cs_index", 0] # Use spi.cs0_gpio  
- ["ili9341.dc_pin", 33]
```

Application

```
# include "mgos.h"
<p class="mume-header" id="include-mgosh-12"></p>

# include "mgos_ili9341.h"
<p class="mume-header" id="include-mgos_ilis341h"></p>

# include "fonts/FreeMonoBold9pt7b.h"
<p class="mume-header" id="include-fontsfreemonobold9pt7bh"></p>

enum mgos_app_init_result mgos_app_init(void) {
    mgos_ilis341_set_fgcolor(0xff, 0, 0);           // Red
    mgos_ilis341_set_window(20, 30, 119, 59);       // 100x30 pixels
    mgos_ilis341_drawRoundRect(0, 0, 100, 30, 8);   // Draw a rounded rectangle
    mgos_ilis341_set_fgcolor(0, 0xff, 0);           // Green
    mgos_ilis341_set_font(&FreeMonoBold9pt7b);       // Set font
    mgos_ilis341_print(5, 5, "Hello World");

    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

This library implements a [analog-to-digital conversion](#) for Mongoose OS.

Blynk integration for Mongoose OS

This Mongoose OS library allows your device remote control via the Blynk platform. Device side logic could be implemented in either C/C++ or JavaScript.

Blynk is a platform with iOS and Android apps to control Arduino, Raspberry Pi and the likes over the Internet.

See example video at:



How to use this library

In your Mongoose OS app, edit `mos.yml` file and add a reference to this library. See an [example blynk app](#) that does that.

Device configuration

This library adds `blynk` configuration section to the device:

```
mos config-get blynk
{
  "auth": "YOUR_BLYNK_AUTH_TOKEN",
  "enable": true,
  "server": "blynk-cloud.com:8442"
}
```

In order for your device to authenticate with Blynk cloud, either use Web UI to change the `blynk.auth` value, or in a terminal:

```
mos config-set blynk.auth=YOUR_BLYNK_AUTH_TOKEN
```

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a

pull request only includes the commits related to that bug or feature.

3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

INA226 I2C Driver

A Mongoose library for Texas Instruments' popular I2C current sensor.

Implementation details

The `INA226` is a simple I2C device that measures a voltage drop over a shunt resistor, as well as the voltage on the bus. It is able to measure voltage up to 36V (in 1.25mV increments) on the bus and the voltage drop over the shunt resistor in 2.5uV increments with a range of 81.92mV, and is very precise.

Using a 0.1Ohm shunt resistor, the maximum current that can be measured is 0.8A.

API Description

Create an `INA226` object using `mgos_ina226_create()`, set the shunt resistor value using `mgos_ina226_set_shunt_resistance()`, then call `mgos_ina226_get_bus_voltage()` to get the bus voltage, call `mgos_ina226_get_shunt_voltage()` to get the voltage drop over the shunt resistor, and `mgos_ina226_get_current()` to get the current flowing through the circuit.

Example application

```

#include "mgos.h"
<p class="mume-header" id="include-mgosh-13"></p>

#include "mgos_config.h"
<p class="mume-header" id="include-mgos_configh-3"></p>

#include "mgos_ina226.h"
<p class="mume-header" id="include-mgos_ina226h"></p>

static void ina226_timer_cb(void *user_data) {
    struct mgos_ina226 *sensor = (struct mgos_ina226 *)user_data;
    float bus, shunt, current, res;

    if (!sensor) return;

    mgos_ina226_get_bus_voltage(sensor, &bus);
    mgos_ina226_get_shunt_resistance(sensor, &res);
    mgos_ina226_get_shunt_voltage(sensor, &shunt);
    mgos_ina226_get_current(sensor, &current);
    LOG(LL_INFO, ("Vbus=% .3f V Vshunt=% .0f uV Rshunt=% .3f Ohm Ishunt=% .1f mA",
        bus, shunt*1e6, res, current*1e3));
}
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_ina226 *sensor;

    sensor = mgos_ina226_create(mgos_i2c_get_global(), mgos_sys_config_get_ina226_i2caddr());
    if (!sensor) {
        LOG(LL_ERROR, ("Could not create INA226 sensor"));
        return false;
    }

    mgos_set_timer(1000, true, ina226_timer_cb, sensor);

    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Crontab

Crontab wraps [cron core](#) and maintains a persisted set of cron jobs. Crontab file is simply a JSON file (actually managed by [jstore](#)) which looks like this:

```
{"items": [
  ["1", {
    "at": "0 0 7 * * MON-FRI",
    "enable": true,
    "action": "foo",
    "payload": {"a": 1, "b": 2}
  }],
  ["2", {
    "at": "0 */2 1-4 * * *",
    "enable": true,
    "action": "bar"
  }]
]}
```

For the cron expression syntax, see [cron core docs](#).

Arduino compatibility for Mongoose OS

This library provides a common Arduino compatibility layer, so that one could pick an existing Arduino program, throw it into the Mongoose OS application sources, and ideally, it "just works".

Currently, the following public headers are provided:

- Arduino.h
- Print.h
- WString.h
- stdlib_noniso.h

There are more specific Arduino-compatibility libraries available: for [onewire](#), [SPI](#), etc.

A Bosch BM222 accelerometer library

Overview

This is a library for the Bosch BM222 accelerometer.

See https://www.bosch-sensortec.com/bst/products/all_products/bma222e for more information about the hardware. VFS subsystem multiplexes calls to libc file API methods such as open, read, write and close between (potentially) several filesystems attached at different mount points.

A filesystem is backed by a device which supports block reads and writes.

Generic MQTT client

This library provides [MQTT protocol](#) client API that allows devices to talk to MQTT servers.

Mongoose OS implements MQTT 3.1.1 client functionality, and works with all popular MQTT server implementations, like AWS IoT, Google IoT Core, Microsoft Azure, IBM Watson, HiveMQ, Mosquitto, etc.

In order to talk to an MQTT server, configure MQTT server settings - see Configuration section below. Once configured, Mongoose OS keeps that connection alive by reconnecting and re-subscribing to all topics after disconnections - you do not need to implement the reconnection logic.

If you want to use TLS, set `mqtt.ssl_ca_cert=ca.pem`. Make sure that `ca.pem` file has required CA certificates. If you want to use mutual TLS, set `mqtt.ssl_cert=CLIENT_CERT.pem` and `mqtt.ssl_key=PRIVATE_KEY.pem`.

See example video (don't forget to set `mqtt.enable=true` before you try it):

Configuration

The MQTT library adds `mqtt` section to the device configuration:

```
{  
  "clean_session": true,           // Clean session info stored on server  
  "client_id": "",                // If not set, device.id is used  
  "enable": false,                 // Enable MQTT functionality  
  "keep_alive": 60,                // How often to send PING messages in seconds  
  "pass": "",                     // User password  
  "reconnect_timeout_min": 2,       // Minimum reconnection timeout in seconds  
  "reconnect_timeout_max": 60,      // Maximum reconnection timeout in seconds  
  "server": "iot.eclipse.org",     // Server to connect to. if `:PORT` is not specified,  
                                // 1883 or 8883 is used depending on whether SSL is enabled.  
  "ssl_ca_cert": "",              // Set this to file name with CA certs to enable TLS  
  "ssl_cert": "",                 // Client certificate for mutual TLS  
  "ssl_cipher_suites": "",         // TLS cipher suites  
  "ssl_key": "",                  // Private key for the client certificate  
  "ssl_psk_identity": "",          // If set, a preshared key auth is used  
  "ssl_psk_key": "",              // Preshared key  
  "user": "",                     // MQTT user name, if MQTT auth is used  
  "will_message": "",              // MQTT last will message  
  "will_topic": ""                // MQTT last will topic  
}
```

Reconnect behavior and backup server

It is possible to have a "backup" server that device will connect to if it fails to connect to the primary server.

Backup server is configured under the `mqtt1` section which contains exactly the same parameters as `mqtt` described above.

Device will first try to connect to the main server configured under `mqtt`.

It will keep connecting to it, increasing the reconnection interval from `reconnect_timeout_min` to `reconnect_timeout_max`.

Reconnection interval is doubled after each attempt so for values above there will be connection attempts after 2, 4, 8, 16, 32 and 60 seconds.

After reaching the maximum reconnect interval and if `mqtt1.enable` is set, it will switch to the `mqtt1` configuration and reset the reconnect interval, so it will try to connect to `mqtt1` the same way.

If that works, it will stay connected to `mqtt1`. If connection drops, it will try to reconnect to `mqtt1` in the same way. If connection to backup server fails, it will go back to the main server and so on.

Arduino OneWire library for Mongoose OS

This library provides an Arduino compatibility layer for onewire by providing an `OneWire.h` public header, so that one could pick an existing Arduino program which uses onewire, throw it into the Mongoose OS application sources, and ideally, it "just works".

Additionally, a mgos-specific API is available, see `include/mgos_arduino_onewire.h` and `mjs_fs/api_arduino_onewire.js`.

DS18B20 Library for the ESP32# Azure IoT Hub support library for Mongoose OS

This library provides [Azure IoT Hub](#) support for Mongoose OS.

Currently only plain MQTT is supported.

See Azure IoT + Mongoose OS tutorial at <https://mongoose-os.com/docs/quickstart/cloud/azure.md>

Authentication

Authentication by both SAS token and X.509 certificate is supported. See the [Authentication section](#) of the documentation for explanation.

`mos azure-iot-setup`

The easiest way to setup Azure cloud connection is by using `mos azure-iot-setup`. Makes sure you have the `az` CLI tool installed, create an IoT Hub, then run:

```
$ mos azure-iot-setup --azure-hub-name MY-HUB-NAME --azure-device-id NEW-DEVICE-ID
```

SAS Token

To use symmetric key authentication, obtain the connection string from the web interface or by using the `az` CLI utility:

```
$ az iot hub device-identity show-connection-string --hub-name my-hub --device-id test1
{
  "cs": "HostName=my-hub.azure-devices.net;DeviceId=test1;SharedAccessKey=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA="
}
```

Enable the Azure client and set the `azure.cs` config setting:

```
$ mos config-set azure.enable=true "azure.cs=HostName=my-hub.azure-devices.net;DeviceId=test1;SharedAccessKey=AAAAAAAAAAAAAAAAAAAAAAA="
```

X.509 Certificate

To use authentication by an X.509 certificate, upload the certificate and private key files in PEM format on the device and configure `azure.host_name`, `azure.device_id`, `azure.cert` and `azure.key`:

```
$ mos put test4.crt.pem  
$ mos put test4.key.pem  
$ mos config-set azure.enable=true azure.host_name=my-hub.azure-devices.net azure.device_id=test4 \  
    azure.cert=test4.crt.pem azure.key=test4.key.pem
```

Note: It is possible to store private key in a cryptochip, such as [ATECC508A](#) (for example, as described [here](#) for Google IoT Core). Just specify `azure.key=ATCA:0` to use private key in slot 0 of the chip. [mos azure-iot-setup] supports ATECC508 key storage - just add `--use-a-tca` to the setup command above.

How to contribute

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various

git tools.

- Finally, push the commits to your fork and submit a [pull request](#).

CCS811 I2C Driver

A Mongoose library for AMS CCS811 integrated circuit.

Sensor details

The CCS811 is an ultra-low power digital gas sensor solution which integrates a metal oxide (MOX) gas sensor to detect a wide range of Volatile Organic Compounds (VOCs) for indoor air quality monitoring with a microcontroller unit (MCU), which includes an Analog-to-Digital converter (ADC), and an I2C interface.

CCS811 is based on ams unique micro-hotplate technology which enables a highly reliable solution for gas sensors, very fast cycle times and a significant reduction in average power consumption.

The integrated MCU manages the sensor driver modes and measurements. The I2C digital interface significantly simplifies the hardware and software design, enabling a faster time to market.

CCS811 supports intelligent algorithms to process raw sensor measurements to output a TVOC value or equivalent CO₂ (eCO₂) levels, where the main cause of VOCs is from humans.

CCS811 supports multiple measurement modes that have been optimized for low-power consumption during an active sensor measurement and idle mode extending battery life in portable applications.

See [datasheet](#)

for implementation details.

A great place to buy a ready made and tested unit is at [Adafruit](#).

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```

#include "mgos.h"
<p class="mume-header" id="include-mgosh-14"></p>

#include "mgos_i2c.h"
<p class="mume-header" id="include-mgos_i2ch"></p>

#include "mgos_ccs811.h"
<p class="mume-header" id="include-mgos_ccs811h"></p>

static struct mgos_ccs811 *s_ccs811;

static void timer_cb(void *user_data) {
    float eco2, tvoc;

    eco2=mgos_ccs811_get_eco2(s_ccs811);
    tvoc=mgos_ccs811_get_tvoc(s_ccs811);

    LOG(LL_INFO, ("ccs811 eCO2=%.0fppm TVOC=%.0fppb", eco2, tvoc));

    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_i2c *i2c;

    i2c=mgos_i2c_get_global();
    if (!i2c) {
        LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
    } else {
        s_ccs811=mgos_ccs811_create(i2c, 0x5a); // Default I2C address
        if (s_ccs811) {
            mgos_set_timer(5000, true, timer_cb, NULL);
        } else {
            LOG(LL_ERROR, ("Could not initialize sensor"));
        }
    }
    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Miniz compression library

Overview

Provides support for reading and writing ZIP archives.

This is the version 2.0.7 of the library, obtained from [here](#).

License

MIT 3-clause:

Copyright 2013-2014 RAD Game Tools and Valve Software
Copyright 2010-2014 Rich Geldreich and Tenacious Software LLC

All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

OTA firmware update via the device shadow

Overview

This library provides a way to update firmware Over-The-Air via the device shadow mechanism.

In the text below, AWS IoT Shadow will be used.

It works by observing the `ota.url` entry in the device shadow. That entry must be an URL with the valid Mongoose OS firmware, which is a .zip file. When that entry is changed, this library downloads a firmware from that URL and kicks off an OTA update. The sequence of actions is as follows:

- Receive shadow delta for `ota.url` - a new `ota.url` value
- Compare `ota.url` with file on flash. If it is the same as new value, stop
- If it is different, save new `ota.url` locally on flash
- Report new `ota_url`
- Trigger an OTA by downloading the .zip
- During the process, report numeric `ota.code` and string `ota.message` entries which indicates the status of the OTA in nearly real time
- On any failure, stop with the failure `ota.message`
- On success, reboot to the new firmware
- After reboot, commit the new firmware after the successful AWS IoT handshake
- If AWS IoT handshake does not happen during 5 minutes (the default commit timeout), rollback to the old firmware
- OTA failure keeps the `ota.url` delta uncleared

How to trigger an OTA firmware update

Build and flash the <https://github.com/mongoose-os-apps/demo-js> app, start the console, and provision to AWS IoT. Open the shadow GUI for the device.

Upload built firmware to the

Mongoose OS dashboard

and copy the firmware URL.

Change the desired shadow state by adding a section `ota` with `url` key:

```
{  
  "desired": {  
    "ota": {  
      "url": "https://mongoose-os.com/dashboard/ota/PATH/TO/APP.zip"  
    },  
    ...  
  },  
  ...  
}
```

You can edit device shadow from the AWS IoT dashboard:

When it successfully finishes, it reboots and commits the firmware after the AWS IoT handshake:

PPPoS / cellular modem support

This library provides IP over serial port. Encapsulation is PPP.

Settings

```
"pppos": {  
  "enable": false,          # Enable PPPoS  
  "uart_no": 1,            # Which UART to use.  
  "baud_rate": 115200,     # Baud rate, data mode is 8-N-1.  
  "fc_enable": false,       # Enable hardware CTS/RTS flow control  
  "apn": "",               # APN name  
  "user": "",               # User name  
  "pass": "",               # Password  
  "connect_cmd": "ATDT*99***1#", # AT command to send to initiate PPP data mode  
  "echo_interval": 10,       # LCP Echo interval, seconds  
  "echo_fails": 3,           # Number of failed echos before connection is declared dead are retried  
  "hexdump_enable": false    # Dump all the data sent over UART to stderr  
}
```

Default UART pin assignments are used and they can be found [here](#).

Example configuration

Access Point Name, PPP username and password depend on the operator. They are usually public and can be found [here](#).

Here's an example for Vodafone Ireland:

```
"pppos": {  
  "enable": true,  
  "uart_no": 1,  
  "apn": "live.vodafone.ie",  
  "user": "dublin",  
  "pass": "dublin",  
}
```

RPC Service - WiFi

Wifi.Scan

Scan wifi networks.

Arguments: none.

Example usage:

```
mos call Wifi.Scan
[
  {
    "ssid": "my_essid",
    "bssid": "12:34:56:78:90:ab",
    "auth": 0,
    "channel": 1,
    "rssi": -25
  },
  ...
]
```

Authentication ENUM mapping

From mgos_wifi.h

```
enum mgos_wifi_auth_mode {
  MGOS_WIFI_AUTH_MODE_OPEN = 0,
  MGOS_WIFI_AUTH_MODE_WEP = 1,
  MGOS_WIFI_AUTH_MODE_WPA_PSK = 2,
  MGOS_WIFI_AUTH_MODE_WPA2_PSK = 3,
  MGOS_WIFI_AUTH_MODE_WPA_WPA2_PSK = 4,
  MGOS_WIFI_AUTH_MODE_WPA2_ENTERPRISE = 5,
};
```

README for git hooks script

git has a way to run scripts, which are invoked by specific git commands.

The git hooks are located in `<mbed TLS root>/.git/hooks`, and as such are not under version control
for more information, see the [git documentation](#).

The mbed TLS git hooks are located in `<mbed TLS root>/tests/git-scripts` directory, and one must create a soft link from `<mbed TLS root>/.git/hooks` to `<mbed TLS root>/tesst/git-scripts`, in order to make the hook scripts successfully work.

Example:

Execute the following command to create a link on linux from the mbed TLS `.git/hooks` directory:

```
In -s ../../tests/git-scripts/pre-push.sh pre-push
```

Note: Currently the mbed TLS git hooks work only on a GNU platform. If using a non-GNU platform, don't enable these hooks!

These scripts can also be used independently.

Note: This is just a template, so feel free to use/remove the unnecessary things

Description

- Type: Bug | Enhancement\Feature Request | Question

- Priority: Blocker | Major | Minor
-

Bug

OS

Mbed OS|linux|windows|

mbed TLS build:

Version: x.x.x or git commit id

OS version: x.x.x

Configuration: please attach config.h file where possible

Compiler and options (if you used a pre-built binary, please indicate how you obtained it):

Additional environment information:

Peer device TLS stack and version

OpenSSL|GnuTLS|Chrome|NSS(Firefox)|SecureChannel (IIS/Internet Explorer/Edge)|Other

Version:

Expected behavior

Actual behavior

Steps to reproduce

Enhancement\Feature Request

Justification - why does the library need this feature?

Suggested enhancement

Question

Please first check for answers in the [Mbed TLS knowledge Base](#), and preferably file an issue in the [Mbed TLS support forum](#)

Notes:

- Pull requests cannot be accepted until:
- The submitter has [accepted the online agreement here](#) with a click through
or for companies or those that do not wish to create an mbed account, a slightly different agreement can be found [here](#)
- The PR follows the [mbed TLS coding standards](#)
- This is just a template, so feel free to use/remove the unnecessary things

Description

A few sentences describing the overall goals of the pull request's commits.

Status

READY/IN DEVELOPMENT/HOLD

Requires Backporting

When there is a bug fix, it should be backported to all maintained and supported branches.

Changes do not have to be backported if:

- This PR is a new feature\enhancement
- This PR contains changes in the API. If this is true, and there is a need for the fix to be backported, the fix should be handled differently in the legacy branch

Yes | NO

Which branch?

Migrations

If there is any API change, what's the incentive and logic for it.

YES | NO

Additional comments

Any additional information that could be of interest

Todos

- [] Tests
- [] Documentation
- [] Changelog updated
- [] Backported

Steps to test or reproduce

Outline the steps to test or reproduce the PR here. Contributing

We gratefully accept bug reports and contributions from the community. There are some requirements we need to fulfill in order to be able to integrate contributions:

- As with any open source project, contributions will be reviewed by the project team and community and may need some modifications to be accepted.
- The contribution should not break API or ABI, unless there is a real justification for that. If there is an API change, the contribution, if accepted, will be merged only when there will be a major release.

Contributor License Agreement (CLA)

- All contributions, whether large or small, require a Contributor's License Agreement (CLA) to be accepted. This is because source code can possibly fall under copyright law and we need your consent to share in the ownership of the copyright.
- To accept the Contributor's License Agreement (CLA), individual contributors can do this by creating an Mbed account and [accepting the online agreement here with a click through](#). Alternatively, for contributions from corporations, or those that do not wish to create an Mbed account, a slightly different agreement can be found [here](#). This agreement should be signed and returned to Arm as described in the instructions given.

Coding Standards

- We would ask that contributions conform to [our coding standards](#), and that contributions are fully tested before submission, as mentioned in the [Tests](#) and [Continuous Integration](#) sections.
- The code should be written in a clean and readable style.
- The code should be written in a portable generic way, that will benefit the whole community, and not only your own needs.
- The code should be secure, and will be reviewed from a security point of view as well.

Making a Contribution

1. [Check for open issues](#) or [start a discussion](#) around a feature idea or a bug.
2. Fork the [Mbed TLS repository on GitHub](#) to start making your changes. As a general rule, you should use the "[development](#)" branch as a basis.
3. Write a test which shows that the bug was fixed or that the feature works as expected.
4. Send a pull request (PR) and work with us until it gets merged and published. Contributions may need some modifications, so a few rounds of review and fixing may be necessary. We will include your name in the ChangeLog 😊
5. For quick merging, the contribution should be short, and concentrated on a single feature or topic. The larger the contribution is, the longer it would take to review it and merge it.
6. Mbed TLS is released under the Apache license, and as such, all the added files should include the Apache license header.

API/ABI Compatibility

The project aims to minimise the impact on users upgrading to newer versions of the library and it should not be necessary for a user to make any changes to their own code to work with a newer version of the library. Unless the user has made an active decision to use newer features, a newer generation of the library or a change has been necessary due to a security issue or other significant software defect, no modifications to their own code should be necessary. To achieve this, API compatibility is maintained between different versions of Mbed TLS on the main development branch and in LTS (Long Term Support) branches.

To minimise such disruption to users, where a change to the interface is required, all changes to the ABI or API, even on the main development branch where new features are added, need to be justifiable by either being a significant enhancement, new feature or bug fix which is best resolved by an interface change.

Where changes to an existing interface are necessary, functions in the public interface which need to be changed, are marked as 'deprecated'. This is done with the preprocessor symbols `MBEDTLS_DEPRECATED_WARNING` and `MBEDTLS_DEPRECATED_REMOVED`. Then, a new function with a new name but similar if not identical behaviour to the original function containing the necessary changes should be created alongside the existing deprecated function.

When a build is made with the deprecation preprocessor symbols defined, a compiler warning will be generated to warn a user that the function will be removed at some point in the future, notifying users that they should change from the older deprecated function to the newer function at their own convenience.

Therefore, no changes are permitted to the definition of functions in the public interface which will change the API. Instead the interface can only be changed by its extension. As described above, if a function needs to be changed, a new function needs to be created alongside it, with a new name, and whatever change is necessary, such as a new parameter or the addition of a return value.

Periodically, the library will remove deprecated functions from the library which will be a breaking change in the API, but such changes will be made only in a planned, structured way that gives sufficient notice to users of the library.

Long Term Support Branches

Mbed TLS maintains several LTS (Long Term Support) branches, which are maintained continuously for a given period. The LTS branches are provided to allow users of the library to have a maintained, stable version of the library which contains only security fixes and fixes for other defects, without encountering additional features or API extensions which may introduce issues or change the code size or RAM usage, which can be significant considerations on some platforms. To allow users to take advantage of the LTS branches, these branches maintain backwards compatibility for both the public API and ABI.

When backporting to these branches please observe the following rules:

1. Any change to the library which changes the API or ABI cannot be backported.
2. All bug fixes that correct a defect that is also present in an LTS branch must be backported to that LTS branch. If a bug fix introduces a change to the API such as a new function, the fix should be reworked to avoid the API change. API changes without very strong justification are unlikely to be accepted.
3. If a contribution is a new feature or enhancement, no backporting is required. Exceptions to this may be additional test cases or quality improvements such as changes to build or test scripts.

It would be highly appreciated if contributions are backported to LTS branches in addition to the [development branch](#) by contributors.

Currently maintained LTS branches are:

1. [mbedtls-2.7](#)
2. [mbedtls-2.16](#)

Tests

As mentioned, tests that show the correctness of the feature or bug fix should be added to the pull request, if no such tests exist.

Mbed TLS includes a comprehensive set of test suites in the `tests/` directory that are dynamically generated to produce the actual test source files (e.g. `test_suite_mpi.c`). These files are generated from a `function file` (e.g. `suites/test_suite_mpi.function`) and a `data file` (e.g. `suites/test_suite_mpi.data`). The function file contains the test functions. The data file contains the test cases, specified as parameters that will be passed to the test function.

A [Knowledge Base article describing how to add additional tests is available on the Mbed TLS website](#).

A test script `tests/scripts/basic-build-test.sh` is available to show test coverage of the library. New code contributions should provide a similar level of code coverage to that which already exists for the library.

Sample applications, if needed, should be modified as well.

Continuous Integration Tests

Once a PR has been made, the Continuous Integration (CI) tests are triggered and run. You should follow the result of the CI tests, and fix failures.

It is advised to enable the [githooks scripts](#) prior to pushing your changes, for catching some of the issues as early as possible.

Documentation

Mbed TLS is well documented, but if you think documentation is needed, speak out!

1. All interfaces should be documented through Doxygen. New APIs should introduce Doxygen documentation.
2. Complex parts in the code should include comments.
3. If needed, a Readme file is advised.
4. If a [Knowledge Base \(KB\)](#) article should be added, write this as a comment in the PR description.
5. A [ChangeLog](#) entry should be added for this contribution.

README for Mbed TLS

Configuration

Mbed TLS should build out of the box on most systems. Some platform specific options are available in the fully documented configuration file `include/mbedtls/config.h`, which is also the place where features can be selected. This file can be edited manually, or in a more

programmatic way using the Perl script `scripts/config.pl` (use `--help` for usage instructions).

Compiler options can be set using conventional environment variables such as `CC` and `CFLAGS` when using the Make and CMake build system (see below).

Compiling

There are currently three active build systems used within Mbed TLS releases:

- GNU Make
- CMake
- Microsoft Visual Studio (Microsoft Visual Studio 2010 or later)

The main systems used for development are CMake and GNU Make. Those systems are always complete and up-to-date. The others should reflect all changes present in the CMake and Make build system, although features may not be ported there automatically.

The Make and CMake build systems create three libraries: `libmbedtls`, `libmbedx509`, and `libmbeddtls`. Note that `libmbeddtls` depends on `libmbedx509` and `libmbedcrypto`, and `libmbedx509` depends on `libmbedcrypto`. As a result, some linkers will expect flags to be in a specific order, for example the GNU linker wants `-Lmbeddtls -Lmbedx509 -Lmbedcrypto`. Also, when loading shared libraries using `dlopen()`, you'll need to load `libmbedcrypto` first, then `libmbedx509`, before you can load `libmbeddtls`.

Make

We require GNU Make. To build the library and the sample programs, GNU Make and a C compiler are sufficient. Some of the more advanced build targets require some Unix/Linux tools.

We intentionally only use a minimum of functionality in the makefiles in order to keep them as simple and independent of different toolchains as possible, to allow users to more easily move between different platforms. Users who need more features are recommended to use CMake.

In order to build from the source code using GNU Make, just enter at the command line:

```
make
```

In order to run the tests, enter:

```
make check
```

The tests need Python to be built and Perl to be run. If you don't have one of them installed, you can skip building the tests with:

```
make no_test
```

You'll still be able to run a much smaller set of tests with:

```
programs/test/selftest
```

In order to build for a Windows platform, you should use `WINDOWS_BUILD=1` if the target is Windows but the build environment is Unix-like (for instance when cross-compiling, or compiling from an MSYS shell), and `WINDOWS=1` if the build environment is a Windows shell (for instance using mingw32-make) (in that case some targets will not be available).

Setting the variable `SHARED` in your environment will build shared libraries in addition to the static libraries. Setting `DEBUG` gives you a debug build. You can override `CFLAGS` and `LDFLAGS` by setting them in your environment or on the make command line; compiler warning options may be overridden separately using `WARNING_CFLAGS`. Some directory-specific options (for example, `-I` directives) are still preserved.

Please note that setting `CFLAGS` overrides its default value of `-O2` and setting `WARNING_CFLAGS` overrides its default value (starting with `-Wall -W`), so if you just want to add some warning options to the default ones, you can do so by setting `CFLAGS=-O2 -Werror` for example.

Setting `WARNING_CFLAGS` is useful when you want to get rid of its default content (for example because your compiler doesn't accept `-Wai` as an option). Directory-specific options cannot be overridden from the command line.

Depending on your platform, you might run into some issues. Please check the Makefiles in `library/`, `programs/` and `tests/` for options to manually add or remove for specific platforms. You can also check [the Mbed TLS Knowledge Base](#) for articles on your platform or issue.

In case you find that you need to do something else as well, please let us know what, so we can add it to the [Mbed TLS Knowledge Base](#).

CMake

In order to build the source using CMake in a separate directory (recommended), just enter at the command line:

```
mkdir /path/to/build_dir && cd /path/to/build_dir  
cmake /path/to/mbedtls_source  
make
```

In order to run the tests, enter:

```
make test
```

The test suites need Python to be built and Perl to be executed. If you don't have one of these installed, you'll want to disable the test suites with:

```
cmake -DENABLE_TESTING=Off /path/to/mbedtls_source
```

If you disabled the test suites, but kept the programs enabled, you can still run a much smaller set of tests with:

```
programs/test/selftest
```

To configure CMake for building shared libraries, use:

```
cmake -DUSE_SHARED_MBEDTLS_LIBRARY=On /path/to/mbedtls_source
```

There are many different build modes available within the CMake buildsystem. Most of them are available for gcc and clang, though some are compiler-specific:

- `Release`. This generates the default code without any unnecessary information in the binary files.
- `Debug`. This generates debug information and disables optimization of the code.
- `Coverage`. This generates code coverage information in addition to debug information.
- `ASan`. This instruments the code with AddressSanitizer to check for memory errors. (This includes LeakSanitizer, with recent version of gcc and clang.) (With recent version of clang, this mode also instruments the code with UndefinedSanitizer to check for undefined behaviour.)
- `ASanDbg`. Same as ASan but slower, with debug information and better stack traces.
- `MemSan`. This instruments the code with MemorySanitizer to check for uninitialized memory reads. Experimental, needs recent clang on Linux/x86_64.
- `MemSanDbg`. Same as MemSan but slower, with debug information, better stack traces and origin tracking.
- `Check`. This activates the compiler warnings that depend on optimization and treats all warnings as errors.

Switching build modes in CMake is simple. For debug mode, enter at the command line:

```
cmake -D CMAKE_BUILD_TYPE=Debug /path/to/mbedtls_source
```

To list other available CMake options, use:

```
cmake -LH
```

Note that, with CMake, you can't adjust the compiler or its flags after the

initial invocation of cmake. This means that `CC=your_cc` make and `make CC=your_cc` will not work (similarly with `CFLAGS` and other variables).

These variables need to be adjusted when invoking cmake for the first time, for example:

```
CC=your_cc cmake /path/to/mbedtls_source
```

If you already invoked cmake and want to change those settings, you need to remove the build directory and create it again.

Note that it is possible to build in-place; this will however overwrite the provided Makefiles (see `scripts/tmp_ignore_makefiles.sh` if you want to prevent `git status` from showing them as modified). In order to do so, from the Mbed TLS source directory, use:

```
cmake .
make
```

If you want to change `CC` or `CFLAGS` afterwards, you will need to remove the CMake cache. This can be done with the following command using GNU find:

```
find . -iname '*cmake*' -not -name CMakeLists.txt -exec rm -rf {} +
```

You can now make the desired change:

```
CC=your_cc cmake .
make
```

Regarding variables, also note that if you set `CFLAGS` when invoking `cmake`, your value of `CFLAGS` doesn't override the content provided by `cmake` (depending on the build mode as seen above), it's merely prepended to it.

Microsoft Visual Studio

The build files for Microsoft Visual Studio are generated for Visual Studio 2010.

The solution file `mbedtls.sln` contains all the basic projects needed to build the library and all the programs. The files in tests are not generated and compiled, as these need Python and perl environments as well. However, the selftest program in `programs/test/` is still available.

Example programs

We've included example programs for a lot of different features and uses in `programs/`. Most programs only focus on a single feature or usage scenario, so keep that in mind when copying parts of the code.

Tests

Mbed TLS includes an elaborate test suite in `tests/` that initially requires Python to generate the tests files (e.g. `test_suite\mpi.c`). These files are generated from a `function file` (e.g. `suites/test_suite\mpi.function`) and a `data file` (e.g. `suites/test_suite\mpi.data`). The `function file` contains the test functions. The `data file` contains the test cases, specified as parameters that will be passed to the test function.

For machines with a Unix shell and OpenSSL (and optionally GnuTLS) installed, additional test scripts are available:

- `tests/ssl-opt.sh` runs integration tests for various TLS options (renegotiation, resumption, etc.) and tests interoperability of these options with other implementations.
- `tests/compat.sh` tests interoperability of every ciphersuite with other implementations.

- `tests/scripts/test-ref-configs.pl` test builds in various reduced configurations.
- `tests/scripts/key-exchanges.pl` test builds in configurations with a single key exchange enabled
- `tests/scripts/all.sh` runs a combination of the above tests, plus some more, with various build options (such as ASan, full `config.h`, etc).

Configurations

We provide some non-standard configurations focused on specific use cases in the `configs/` directory. You can read more about those in `configs/README.txt`

Porting Mbed TLS

Mbed TLS can be ported to many different architectures, OS's and platforms. Before starting a port, you may find the following Knowledge Base articles useful:

- [Porting Mbed TLS to a new environment or OS](#)
- [What external dependencies does Mbed TLS rely on?](#)
- [How do I configure Mbed TLS](#)

Contributing

We gratefully accept bug reports and contributions from the community. There are some requirements we need to fulfill in order to be able to integrate contributions:

- All contributions, whether large or small require a Contributor's License Agreement (CLA) to be accepted. This is because source code can possibly fall under copyright law and we need your consent to share in the ownership of the copyright.
- We would ask that contributions conform to [our coding standards](#), and that contributions should be fully tested before submission.
- As with any open source project, contributions will be reviewed by the project team and community and may need some modifications to be accepted.

To accept the Contributor's Licence Agreement (CLA), individual contributors can do this by creating an Mbed account and [accepting the online agreement here with a click through](#). Alternatively, for contributions from corporations, or those that do not wish to create an Mbed account, a slightly different agreement can be found [here](#). This agreement should be signed and returned to Arm as described in the instructions given.

Making a Contribution

1. [Check for open issues](#) or [start a discussion](#) around a feature idea or a bug.
2. Fork the [Mbed TLS repository on GitHub](#) to start making your changes. As a general rule, you should use the "development" branch as a basis.
3. Write a test which shows that the bug was fixed or that the feature works as expected.
4. Send a pull request and bug us until it gets merged and published. Contributions may need some modifications, so work with us to get your change accepted. We will include your name in the ChangeLog 😊

Mbed TLS sample programs

This subdirectory mostly contains sample programs that illustrate specific features of the library, as well as a few test and support programs.

Symmetric cryptography (AES) examples

- `aes/aescrypt2.c`: file encryption and authentication with a key derived from a low-entropy secret, demonstrating the low-level AES interface, the digest interface and HMAC.

Warning: this program illustrates how to use low-level functions in the library. It should not be taken as an example of how to build a secure encryption mechanism. To derive a key from a low-entropy secret such as a password, use a standard key stretching mechanism such as PBKDF2 (provided by the `pkcs5` module). To encrypt and authenticate data, use a standard mode such as GCM or CCM (both available as library module).

- `aes/crypt_and_hash.c`: file encryption and authentication, demonstrating the generic cipher interface and the generic hash interface.

Hash (digest) examples

- `hash/generic_sum.c`: file hash calculator and verifier, demonstrating the message digest (`md`) interface.
- `hash/hello.c`: hello-world program for MD5.

Public-key cryptography examples

Generic public-key cryptography (`pk`) examples

- `pkey/gen_key.c`: generates a key for any of the supported public-key algorithms (RSA or ECC) and writes it to a file that can be used by the other pk sample programs.
- `pkey/key_app.c`: loads a PEM or DER public key or private key file and dumps its content.
- `pkey/key_app_writer.c`: loads a PEM or DER public key or private key file and writes it to a new PEM or DER file.
- `pkey/pk_encrypt.c`, `pkey/pk_decrypt.c`: loads a PEM or DER public/private key file and uses the key to encrypt/decrypt a short string through the generic public-key interface.
- `pkey/pk_sign.c`, `pkey/pk_verify.c`: loads a PEM or DER private/public key file and uses the key to sign/verify a short string.

ECDSA and RSA signature examples

- `pkey/ecdsa.c`: generates an ECDSA key, signs a fixed message and verifies the signature.
- `pkey/rsa_encrypt.c`, `pkey/rsa_decrypt.c`: loads an RSA public/private key and uses it to encrypt/decrypt a short string through the low-level RSA interface.
- `pkey/rsa_genkey.c`: generates an RSA key and writes it to a file that can be used with the other RSA sample programs.
- `pkey/rsa_sign.c`, `pkey/rsa_verify.c`: loads an RSA private/public key and uses it to sign/verify a short string with the RSA PKCS#1 v1.5 algorithm.
- `pkey/rsa_sign_pss.c`, `pkey/rsa_verify_pss.c`: loads an RSA private/public key and uses it to sign/verify a short string with the RSASSA-PSS algorithm.

Diffie-Hellman key exchange examples

- `pkey/dh_client.c`, `pkey/dh_server.c`: secure channel demonstrators (client, server). This pair of programs illustrates how to set up a secure channel using RSA for authentication and Diffie-Hellman to generate a shared AES session key.
- `pkey/ecdh_curve25519.c`: demonstration of a elliptic curve Diffie-Hellman (ECDH) key agreement.

Bignum (`mpi`) usage examples

- `pkey/dh_genprime.c`: shows how to use the bignum (`mpi`) interface to generate Diffie-Hellman parameters.
- `pkey/mpi_demo.c`: demonstrates operations on big integers.

Random number generator (RNG) examples

- `random/gen_entropy.c`: shows how to use the default entropy sources to generate random data.
Note: most applications should only use the entropy generator to seed a cryptographic pseudorandom generator, as illustrated by `random/gen_random_ctr_drbg.c`.
- `random/gen_random_ctr_drbg.c`: shows how to use the default entropy sources to seed a pseudorandom generator, and how to use the resulting random generator to generate random data.
- `random/gen_random_havege.c`: demonstrates the HAVEGE entropy collector.

SSL/TLS examples

SSL/TLS sample applications

- `ssl/dtls_client.c`: a simple DTLS client program, which sends one datagram to the server and reads one datagram in response.
- `ssl/dtls_server.c`: a simple DTLS server program, which expects one datagram from the client and writes one datagram in response. This program supports DTLS cookies for hello verification.
- `ssl/mini_client.c`: a minimalistic SSL client, which sends a short string and disconnects. This is primarily intended as a benchmark; for a better example of a typical TLS client, see `ssl/ssl_client1.c`.
- `ssl/ssl_client1.c`: a simple HTTPS client that sends a fixed request and displays the response.
- `ssl/ssl_fork_server.c`: a simple HTTPS server using one process per client to send a fixed response. This program requires a Unix/POSIX environment implementing the `fork` system call.
- `ssl/ssl_mail_client.c`: a simple SMTP-over-TLS or SMTP-STARTTLS client. This client sends an email with fixed content.
- `ssl/ssl_pthread_server.c`: a simple HTTPS server using one thread per client to send a fixed response. This program requires the `pthread` library.
- `ssl/ssl_server.c`: a simple HTTPS server that sends a fixed response. It serves a single client at a time.

SSL/TLS feature demonstrators

Note: unlike most of the other programs under the `programs/` directory, these two programs are not intended as a basis for writing an application. They combine most of the features supported by the library, and most applications require only a few features. To write a new application, we recommended that you start with `ssl_client1.c` or `ssl_server.c`, and then look inside `ssl/ssl_client2.c` or `ssl/ssl_server2.c` to see how to use the specific features that your application needs.

- `ssl/ssl_client2.c`: an HTTPS client that sends a fixed request and displays the response, with options to select TLS protocol features and Mbed TLS library features.
- `ssl/ssl_server2.c`: an HTTPS server that sends a fixed response, with options to select TLS protocol features and Mbed TLS library features.

In addition to providing options for testing client-side features, the `ssl_client2` program has options that allow you to trigger certain behaviors in the server. For example, there are options to select ciphersuites, or to force a renegotiation. These options are useful for testing the corresponding features in a TLS server. Likewise, `ssl_server2` has options to activate certain behaviors that are useful for testing a TLS client.

Test utilities

- `test/benchmark.c`: benchmark for cryptographic algorithms.
- `test/selftest.c`: runs the self-test function in each library module.
- `test/udp_proxy.c`: a UDP proxy that can inject certain failures (delay, duplicate, drop). Useful for testing DTLS.
- `test/zeroize.c`: a test program for `mbedtls_platform_zeroize`, used by `tests/scripts/test_zeroize.gdb`.

Development utilities

- `util/pem2der.c`: a PEM to DER converter. Mbed TLS can read PEM files directly, but this utility can be useful for interacting with other tools or with minimal Mbed TLS builds that lack PEM support.
- `util/strerror.c`: prints the error description corresponding to an integer status returned by an Mbed TLS function.

X.509 certificate examples

- `x509/cert_app.c`: connects to a TLS server and verifies its certificate chain.
- `x509/cert_req.c`: generates a certificate signing request (CSR) for a private key.
- `x509/cert_write.c`: signs a certificate signing request, or self-signs a certificate.
- `x509/crl_app.c`: loads and dumps a certificate revocation list (CRL).
- `x509/req_app.c`: loads and dumps a certificate signing request (CSR).

mbedTLS library for Mongoose OS

This library is optimized by Cesanta for lower memory usage.

It also includes support for ATECC508 crypto chip.

GATT Client RPC service

Provides [RPC service](#) for

[GATT](#)

client.

Methods

- `GATTC.Scan` - performs BLE scan of the area. Results are returned as an array of objects containing address, name (if present) and RSSI.

```
$ mos call GATTC.Scan
{
  "results": [
    {
      "addr": "eb:12:dd:51:19:3d",
      "rssi": -48
    },
    {
      "addr": "24:0a:c4:00:31:be",
      "name": "esp32_0031BC",
      "rssi": -36
    }
  ]
}
```

- `GATTC.Open` - open connection to a device. Device can be specified by either `addr` or `name`. Optional `mtu` parameter specifies link MTU to be used, the default is 23 bytes (as per standard). Returned value is `conn_id` which is the connection identifier to be used for later operations.

```
$ mos call GATTC.Open '{"addr": "eb:12:dd:51:19:3d"}'
{
  "conn_id": 0
}
$ mos call GATTC.Open '{"name": "esp32_0031BC", "mtu": 200}'
{
  "conn_id": 1
}
```

- `GATTS.ListServices` - list services provided by device. For each discovered service, its UUID, instance number and primary flag are returned.

```
$ mos call GATTC.ListServices '{"conn_id": 1}'
{
  "results": [
    {
      "uuid": "1801",
      "instance": 0,
      "primary": true
    },
    {
      "uuid": "1800",
      "instance": 0,
      "primary": true
    },
    {
      "uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f",
      "instance": 0,
      "primary": true
    },
    {
      "uuid": "5f6d4f53-5f52-5043-5f53-56435f49445f",
      "instance": 0,
      "primary": true
    }
  ]
}
```

In this example, we see two standard services - [Generic Access](#) (1800) and [Generic Attribute](#), followed by custom mOS [configuration](#) and [RPC over GATT](#) services.

- [GATTS.ListCharacteristics](#) - list characteristics of a service. For each characteristic, its UUID and properties are returned.

Properties are returned a string of up to 8 characters:

- R - read
- W - write
- w - write with no response required
- N - notify
- I - indicate
- B - broadcast
- E - extended
- A - auth

```
$ mos call GATTC.ListCharacteristics '{"conn_id": 1, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f"}'
{
  "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f",
  "results": [
    {
      "uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30",
      "props": "W"
    },
    {
      "uuid": "316d4f53-5f43-4647-5f76-616c75655f31",
      "props": "WR"
    },
    {
      "uuid": "326d4f53-5f43-4647-5f73-6176655f5f32",
      "props": "W"
    }
  ]
}
```

- [GATTC.Write](#) - write a value to the specified characteristic. `conn_id`, `svc_uuid` and `char_uuid` specify the characteristic, value can be provided either as plain text `value` or hex-encoded `value_hex` keys. The following two calls are equivalent

```
$ mos call GATT.C.Write '{"conn_id": 1, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30", "value": null}
$ mos call GATT.C.Write '{"conn_id": 1, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30", "value": null}
```

- GATT.C.Read - read avlue of a characteristic `conn_id`, `svc_uuid` and `char_uuid` specify the characteristic, value will be returned as either `value` or `value_hex`: if value is a string consisting of printable charaters that can be represented as a valid JSON string, result will be in the `value` key. Otherwise it will be hexified and sent in `value_hex`.

```
$ mos call GATT.C.Read '{"conn_id": 1, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "316d4f53-5f43-4647-5f76-616c75655f31"}'
{
  "value": "true"
}
```

- GATT.C.Close - close the specified connection.

```
$ mos call GATT.C.Close '{"conn_id": 1}'
null
```

Provisioning one device using another

As a complete example, here is full Bluetooth provisioning process described in [this blog post](#):

```
$ mos call GATT.C.Open '{"name": "esp32_0031BC", "mtu": 200}'
{
  "conn_id": 0
}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "316d4f53-5f43-4647-5f76-616c75655f31", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "326d4f53-5f43-4647-5f73-6176655f5f32", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "316d4f53-5f43-4647-5f76-616c75655f31", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "326d4f53-5f43-4647-5f73-6176655f5f32", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "316d4f53-5f43-4647-5f76-616c75655f31", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "326d4f53-5f43-4647-5f73-6176655f5f32", "value": null}
```

Arduino DS3231 High Precision Real Time Clock (RTC) library for Mongoose OS

Tested and works on esp8266/esp32

Description

DS3231 is a low-cost, extremely accurate I²C real-time clock (RTC), with an integrated temperature-compensated crystal oscillator (TCXO) and crystal. The device incorporates a battery input, disconnect the main power supply and maintains accurate timekeeping. Integrated oscillator improve long-term accuracy of the device and reduces the number of components of the production line.

RTC maintains seconds, minutes, hours, day, date, month, and year information. Less than 31 days of the month, the end date will be automatically adjusted, including corrections for leap year. The clock operates in either the 24 hours or band / AM / PM indication of the 12-hour format. Provides two configurable alarm clock and a calendar can be set to a square wave output. Address and data are transferred serially through an I2C bidirectional bus. See [product page link](#) for more information about the hardware.

Features

- Battery
- Operating voltage: 3.3 - 5.5V
- Clock chip: High-precision clock chip DS3231
- Clock Accuracy: 0-40 °C range, the accuracy 2ppm, the error was about 1 minute
- Calendar alarm clock with two
- Programmable square-wave output
- Real time clock generator seconds, minutes, hours, day, date, month and year timing and provide valid until the year 2100 leap year compensation

Usage

mos.yml, add:

```
config_schema:  
- ["i2c.enable", true]  
libs:  
- origin: https://github.com/mongoose-os-libs/arduino-ds3231
```

init.js, add:

```
load('api_ds3231.js');
```

[\(JavaScript\) usage example](#)

```
#include "mgos_arduino_DS3231.h"
```

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted

one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

SHT31 I2C Driver

A Mongoose library for Sensirion's SHT31 integrated circuit.

Sensor details

SHT3x-DIS is the next generation of Sensirion's temperature and humidity sensors. It builds on a new CMOSens sensor chip that is at the heart of Sensirion's new humidity and temperature platform. The SHT3x-DIS has increased intelligence, reliability and improved accuracy specifications compared to its predecessor. Its functionality includes enhanced signal processing, two distinctive and user selectable I2C addresses and communication speeds of up to 1 MHz. The DFN package has a footprint of 2.5 x 2.5 mm while keeping a height of 0.9 mm. This allows for integration of the SHT3x-DIS into a great variety of applications.

Additionally, the wide supply voltage range of 2.4 to 5.5 V guarantees compatibility with diverse assembly situations. All in all, the SHT3x-DIS incorporates 15 years of knowledge of Sensirion, the leader in the humidity sensor industry.

See [datasheet](#)

for implementation details.

A great place to buy a ready made and tested unit is at [Adafruit](#).

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```
#include "mgos.h"
#include "mgos_i2c.h"
#include "mgos_sht31.h"

static struct mgos_sht31 *s_sht31;

static void timer_cb(void *user_data) {
    float temperature, humidity;

    temperature=mgos_sht31_getTemperature(s_sht31);
    humidity=mgos_sht31_getHumidity(s_sht31);

    LOG(LL_INFO, ("sht31 temperature=%.2f humidity=%.2f", temperature, humidity));

    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_i2c *i2c;

    i2c=mgos_i2c_get_global();
    if (!i2c) {
        LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
    } else {
        s_sht31=mgos_sht31_create(i2c, 0x40); // Default I2C address
        if (s_sht31) {
            mgos_set_timer(5000, true, timer_cb, NULL);
        } else {
            LOG(LL_ERROR, ("Could not initialize sensor"));
        }
    }
    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

SPI Flash VFS Driver

This driver supports external SPI flash chips and makes them available for use with the mOS VFS subsystem.

It add an `spi_flash` device type.

Supported options (`dev_opts`) are:

- `freq` - SPI bus frequency.
- `cs` - SPI bus CS line to use, see the [SPI example](#) for explanation.
- `mode` - SPI mode, 0 - 2.
- `size` - specify size explicitly. If not specified, an attempt is made to detect by querying SDFP data or JEDEC ID. Most modern chips support at least one of these, so specifying size is usually not necessary.
- `wip_mask` - bit mask to apply to status register to determine when the write is in progress (chip is busy). Most chips have bit 0 as the `WIP` bit, and the corresponding mask value is 1. This is the default.

Example of console output when this driver is used:

```
[Jul 28 14:35:20.697] mgos_vfs_dev_open spi_flash ({ "freq": 80000000, "cs": 0 }) -> 0x3ffb47f0
[Jul 28 14:35:20.704] mgos_vfs_dev_spi_fla Chip ID: 20 71, size: 1048576
[Jul 28 14:35:20.714] mgos_vfs_mkfts Create SPIFFS (dev 0x3ffb47f0, opts {"size": 262144, "force": true})
```

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition,

"control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and

may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Apple HomeKit ADK for Mongoose OS

This is a port of the official [Apple HomeKit ADK](#) to Mongoose OS.

Targeted platforms are ESP8266 and ESP32 for now.

Examples

Example Mongoose OS app that uses this library can be found [here](#).

TODO

- MFi chip support
- BLE transport support on ESP32
- Buffer size optimization (<https://github.com/apple/HomeKitADK/issues/13>).

HomeKit ADK Platform Abstraction Layer (PAL)

This document describes the ADK PAL and how to port the ADK to a new platform.

Supported platforms

The ADK ships with existing implementations for the following platforms and cryptographic libraries:

Operating systems and runtime environments

- Darwin (macOS)
- Linux

- OpenSSL (1.1.1c or later)
- MbedTLS (2.18.0 or later)

Additional platforms can be supported by providing a custom PAL implementation.

Design considerations

The PAL abstracts all relevant aspects of the underlying platform. Currently the PAL attempts to avoid memory allocations. This is not necessarily a design goal we'll maintain going forward. The PAL has 3 build types: Test, Debug, and Release. Test uses the Mock PAL and is used for unit tests. Debug and Release use platform-specific backends.

Configuration and weak symbols

Past versions of the ADK used platform-specific constructors to pass options to PAL constructors. The current version of the ADK instead uses a consistent constructor across all platforms. Our goal is for applications to be portable across platforms and across BLE and IP.

Crypto PAL

Ed25519

EdDSA (Edwards-curve Digital Signature Algorithm) signature scheme using SHA-512 and Curve25519.

MbedTLS does not natively support Ed25519. We ship a performant software-only implementation along with our MbedTLS implementation but we strongly encourage silicon vendors to substitute their own production quality implementation (or hardware accelerated implementation).

To disable our implementation we recommend including the C file of our bindings in a new C file and defining `HAVE_CUSTOM_ED25519`, which disables our implementation.

```
#define ED25519_PUBLIC_KEY_BYTES 32
#define ED25519_SECRET_KEY_BYTES 32
#define ED25519_BYTES 64

void HAP_ed25519_public_key(uint8_t pk[ED25519_PUBLIC_KEY_BYTES],
                           const uint8_t sk[ED25519_SECRET_KEY_BYTES]);
void HAP_ed25519_sign(uint8_t sig[ED25519_BYTES],
                      const uint8_t *m, size_t m_len,
                      const uint8_t sk[ED25519_SECRET_KEY_BYTES],
                      const uint8_t pk[ED25519_PUBLIC_KEY_BYTES]);
int HAP_ed25519_verify(const uint8_t sig[ED25519_BYTES],
                       const uint8_t *m, size_t m_len,
                       const uint8_t pk[ED25519_PUBLIC_KEY_BYTES]);
```

X25519

X25519 is an elliptic curve Diffie-Hellman key exchange scheme using Curve25519.

```
#define X25519_SCALAR_BYTES 32
#define X25519_BYTES 32

void HAP_X25519_scalarmult_base(uint8_t r[X25519_BYTES],
                                const uint8_t n[X25519_SCALAR_BYTES]);
void HAP_X25519_scalarmult(uint8_t r[X25519_BYTES],
                           const uint8_t n[X25519_SCALAR_BYTES],
                           const uint8_t p[X25519_BYTES]);
```

ChaCha20 Poly1305

Authenticated Encryption using ChaCha20 stream cipher and Poly1305 authenticator.

```
#define CHACHA20_POLY1305_KEY_BYTES 32
#define CHACHA20_POLY1305_NONCE_BYTES_MAX 12
#define CHACHA20_POLY1305_TAG_BYTES 16

void HAP_chacha20_poly1305_init(HAP_chacha20_poly1305_ctx *ctx,
                                 const uint8_t *n, size_t n_len,
                                 const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
void HAP_chacha20_poly1305_update_enc(HAP_chacha20_poly1305_ctx *ctx,
                                       uint8_t *c,
                                       const uint8_t *m, size_t m_len,
                                       const uint8_t *n, size_t n_len,
                                       const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
void HAP_chacha20_poly1305_update_enc_aad(HAP_chacha20_poly1305_ctx *ctx,
                                           const uint8_t *a, size_t a_len,
                                           const uint8_t *n, size_t n_len,
                                           const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
void HAP_chacha20_poly1305_final_enc(HAP_chacha20_poly1305_ctx *ctx,
                                      uint8_t tag[CHACHA20_POLY1305_TAG_BYTES]);
void HAP_chacha20_poly1305_update_dec(HAP_chacha20_poly1305_ctx *ctx,
                                       uint8_t *m,
                                       const uint8_t *c, size_t c_len,
                                       const uint8_t *n, size_t n_len,
                                       const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
void HAP_chacha20_poly1305_update_dec_aad(HAP_chacha20_poly1305_ctx *ctx,
                                           const uint8_t *a, size_t a_len,
                                           const uint8_t *n, size_t n_len,
                                           const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
int HAP_chacha20_poly1305_final_dec(HAP_chacha20_poly1305_ctx *ctx,
                                     const uint8_t tag[CHACHA20_POLY1305_TAG_BYTES]);
```

Note: *The implementation must support overlapping buffers (m and c).*

If HAVE_CUSTOM_SINGLE_SHOT_CHACHA20_POLY1305 is not set, we synthesize a single shot API from the streaming API above. Otherwise the backend must provide the following API. This is recommended for BLE-only crypto backends. *The streaming API is needed by IP accessories only.*

```
void HAP_chacha20_poly1305_encrypt_aad(uint8_t tag[CHACHA20_POLY1305_TAG_BYTES],
                                         uint8_t *c,
                                         const uint8_t *m, size_t m_len,
                                         const uint8_t *a, size_t a_len,
                                         const uint8_t *n, size_t n_len,
                                         const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);

int HAP_chacha20_poly1305_decrypt_aad(const uint8_t tag[CHACHA20_POLY1305_TAG_BYTES],
                                       uint8_t *m,
                                       const uint8_t *c, size_t c_len,
                                       const uint8_t *a, size_t a_len,
                                       const uint8_t *n, size_t n_len,
                                       const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
```

Note: a_len might be NULL.

Secure Remote Password protocol (SRP6a), an augmented password-authenticated key agreement (PAKE) protocol.

Both OpenSSL and MbedTLS do not natively support SRP6a. We are shipping an SRP6a implementation with our code that uses the BIGNUM support in OpenSSL and MbedTLS respectively. We strongly recommend substituting an optimized implementation for MCU-class silicon.

To replace our implementation we recommend including the C file of our bindings in a new C file and defining *CUSTOM_SRP*, which disables our implementation.

```
#define SRP_PRIME_BYTES 384
#define SRP_SALT_BYTES 16
#define SRP_VERIFIER_BYTES 384
#define SRP_SECRET_KEY_BYTES 32
#define SRP_PUBLIC_KEY_BYTES 384
#define SRP_SCRAMBLING_PARAMETER_BYTES 64
#define SRP_PREMASTER_SECRET_BYTES 384
#define SRP_SESSION_KEY_BYTES 64
#define SRP_PROOF_BYTES 64

void HAP_srp_verifier(uint8_t v[SRP_VERIFIER_BYTES],
                      const uint8_t salt[SRP_SALT_BYTES],
                      const uint8_t *user, size_t user_len,
                      const uint8_t *pass, size_t pass_len);
void HAP_srp_public_key(uint8_t pub_b[SRP_PUBLIC_KEY_BYTES],
                       const uint8_t priv_b[SRP_SECRET_KEY_BYTES],
                       const uint8_t v[SRP_VERIFIER_BYTES]);
void HAP_srp_scrambling_parameter(uint8_t u[SRP_SCRAMBLING_PARAMETER_BYTES],
                                  const uint8_t pub_a[SRP_PUBLIC_KEY_BYTES],
                                  const uint8_t pub_b[SRP_PUBLIC_KEY_BYTES]);
int HAP_srp_premaster_secret(uint8_t s[SRP_PREMASTER_SECRET_BYTES],
                            const uint8_t pub_a[SRP_PUBLIC_KEY_BYTES],
                            const uint8_t priv_b[SRP_SECRET_KEY_BYTES],
                            const uint8_t u[SRP_SCRAMBLING_PARAMETER_BYTES],
                            const uint8_t v[SRP_VERIFIER_BYTES]);
void HAP_srp_session_key(uint8_t k[SRP_SESSION_KEY_BYTES],
                        const uint8_t s[SRP_PREMASTER_SECRET_BYTES]);
void HAP_srp_proof_m1(uint8_t m1[SRP_PROOF_BYTES],
                      const uint8_t *user, size_t user_len,
                      const uint8_t salt[SRP_SALT_BYTES],
                      const uint8_t pub_a[SRP_PUBLIC_KEY_BYTES],
                      const uint8_t pub_b[SRP_PUBLIC_KEY_BYTES],
                      const uint8_t k[SRP_SESSION_KEY_BYTES]);
void HAP_srp_proof_m2(uint8_t m2[SRP_PROOF_BYTES],
                      const uint8_t pub_a[SRP_PUBLIC_KEY_BYTES],
                      const uint8_t m1[SRP_PROOF_BYTES],
                      const uint8_t k[SRP_SESSION_KEY_BYTES]);
```

SHA

Secure Hash Algorithms (SHA1, SHA-256, and SHA-512)

```
#define SHA1_BYTES 20

void HAP_sha1(uint8_t md[SHA1_BYTES], const uint8_t *data, size_t size);

#define SHA256_BYTES 32

void HAP_sha256(uint8_t md[SHA256_BYTES], const uint8_t *data, size_t size);

#define SHA512_BYTES 64

void HAP_sha512(uint8_t md[SHA512_BYTES], const uint8_t *data, size_t size);
```

HMAC SHA1

Hash-based message authentication code using SHA1. *Only needed for IP Accessories.*

```
#define HMAC_SHA1_BYTES SHA1_BYTES

void HAP_hmac_sha1_aad(uint8_t r[HMAC_SHA1_BYTES],
    const uint8_t* key, size_t key_len,
    const uint8_t* in, size_t in_len,
    const uint8_t* aad, size_t aad_len);
```

HKDF-SHA512

HMAC-based key derivation function using SHA-512.

```
void HAP_hkdf_sha512(uint8_t* r, size_t r_len,
    const uint8_t* key, size_t key_len,
    const uint8_t* salt, size_t salt_len,
    const uint8_t* info, size_t info_len);
```

Brute-force resistant password-based key derivation function.

PBKDF2-SHA1

```
void HAP_pbkdf2_hmac_sha1(uint8_t *key, size_t key_len,
    const uint8_t *password, size_t password_len,
    const uint8_t *salt, size_t salt_len,
    uint32_t count);
```

AES-CTR

AES block cipher in CTR mode. *Only needed for IP Accessories.*

```
void HAP_aes_ctr_init(HAP_aes_ctr_ctx *ctx, const uint8_t *key, int size, const uint8_t iv[16]);
void HAP_aes_ctr_encrypt(HAP_aes_ctr_ctx *ctx, uint8_t* ct, const uint8_t* pt, size_t pt_len);
void HAP_aes_ctr_decrypt(HAP_aes_ctr_ctx *ctx, uint8_t* pt, const uint8_t* ct, size_t ct_len);
void HAP_aes_ctr_done(HAP_aes_ctr_ctx *ctx);
```

Debugging on Darwin using visual debugger

Using Visual Studio Code

Installation

- Download VSCode: <https://code.visualstudio.com/>
- To install c/c++ language support: within VSCode press ⌘+p and enter

```
ext install ms-vscode.cpptools
```

- To install debugger: within VSCode, press ⌘+p and enter

```
ext install vadimcn.vscode-lldb
```

- Note: an alternative debugger is necessary because llvm no longer includes lldb-mi which the default debugger uses.

Running an application

- Create a workspace which points to the root of your ADK folder.
- Click "Debug" > "Open Configuration", which will open the launch.json file.

- Set "type" to "lldb"
- Set "program" to the application you want to run.

Example launch.json file for Lightbulb app:

```
{  
  "version": "0.2.0",  
  "configurations": [  
    {  
      "type": "lldb",  
      "request": "launch",  
      "name": "Debug",  
      "program": "${workspaceFolder}/Output/Darwin-x86_64-apple-darwin19.0.0/Debug/IP/Applications/Lightbulb.OpenSSL",  
      "args": [],  
      "cwd": "${workspaceFolder}"  
    }  
  ]  
}
```

Development Environment

Development of HomeKit ADK is supported on the following platforms:

Darwin

Prerequisites

Download and install [Xcode 11](#)

```
brew install openssl@1.1  
brew install mbedtls --HEAD  
brew install wget  
brew install qemu  
brew cask install docker
```

Run docker (Look in Spotlight/Applications folder). This is a one time instruction.

Make sure you go to Docker→Preferences→General and check the option → Start Docker Desktop when you log in

Linux

Prerequisites

```
sudo apt install docker
```

Code Style

Please use the following tools to auto-format your code before submitting a Pull Request.

Linting source code

This project uses `clang-format` tool to lint and format the code.

```
./Tools/linters/clint.sh -h
```

Linting shell scripts

Guidelines

Sample Application

- In DB.c, the following IIDs should be used:
 - 0x...0 Service IID.
 - 0x...1 Service Signature characteristic IID. Only present when the service has linked service or has properties.
 - 0x...2 Name characteristic IID. Services can have a Name characteristic if they are user visible.
 - 0x...3+ Regular characteristics, ordered the same way as in the service specification, with required characteristics listed before optional characteristics.

When updating a DB file for a sample that has already been released, try to avoid changing existing IIDs.

Error Handling

- Unexpected errors `HAPError` are used for exceptional circumstances such as communication failure with I2C, or access problems with key-value store. Those errors should be modeled as `HAPError` and are in most cases bubbled out to the caller until the next error handler.
- Expected errors are conditions such as a key that is not available in the key-value store, or a Wi-Fi configuration that has not yet been set. Such errors should be modeled as separate out parameters, such as `bool *found`, `bool *isSet`, `bool *isDefined`, `bool *isAvailable`, or `bool *isConfigured`. Do not use `bool *isValid` to denote existence of a value.
- The idea is that a caller of a function in general should always be able to treat a `HAPError` as a boolean, and only has to handle expected errors returned through the separate out parameters.
- When a `HAPError` is returned, the caller shall not make assumptions about the validity of any out-parameters.
- When `kHAPError_None` is returned, all applicable out parameters of the function must contain valid values.

Specification References

- Always reference the spec if possible, and always use the exact same format to simplify "Find in Files" searches.
 - First line: `@see` or See followed by exact PDF Title as shown in title bar when opening it.
 - Second line: Section + Section number + exact Section Title.
 - Tables may be referenced as well, in that case use Table instead of Section.
 - Section references are preferred as tables may be harder to find.
- For code comments, quote exactly what part of the spec is referenced.
- Always use the most recent specification version where the feature is documented. If a feature becomes obsolete, use the Doxygen `@obsolete` tag to specify the first revision it got removed. Avoid actually deleting enum values etc, use `@obsolete` instead.

Example for documentation block

```
/**  
 * ...  
 *  
 * @see HomeKit Accessory Protocol Specification R15  
 *   Section 1.2.3.4 Some Sample Section  
 */
```

Example for code comment

```
// ...  
// See HomeKit Accessory Protocol Specification R15  
// Section 1.2.3.4 Some Sample Section
```

Definitions

- Definition name: `kHAPCategory_PascalCase`
- Always explicitly specify the type of the definition, e.g., `uint8_t`, `size_t`.
- Consider using enumerations instead, where possible.

```
/**  
 * Brief description  
 */  
#define kHAPCategory_PascalCase ((size_t) 0x42)
```

Enumerations

- Enumerations are never used to store out-of-range values. Instead, the underlying type is used, e.g., `uint8_t`. Before casting to an enum, perform proper input validation. When receiving an enum, no thorough input validation is necessary as it is already done.
- Type name: `HAPPascalCase`
- Case name: `kHAPTypeName_PascalCase` Always prefix with lower case `k`.
- If there is a default, uninitialized, 0 case, list it first, but do not set explicitly to 0.
- If there is no default case, set first value to 1 to avoid accidents with `memset 0`. Do not set values to other cases.
- For flags, set the initial case to `1 << 0`, the next to `1 << 1` etc.
- Align documentation on column 37 if possible, otherwise use a full separate Doxygen style comment block. If there are some long and some short cases, consider using the full comment block for all cases. A single-line Doxygen style comment block may be used to condense information. If the enum is defined in an indented scope, columns advance by 4 per indentation.
- In interfaces, always use `HAP_ENUM_BEGIN` / `HAP_ENUM_END` for portability across different compiler settings.

```
/**  
 * Enum type documentation.  
 */  
HAP_ENUM_BEGIN(uint8_t, HAPEnumType) {  
    kHAPEnumType_PascalCase1,      /**< Case 1 documentation. */  
    kHAPEnumType_PascalCase2,      /**< Case 2 documentation. */  
  
    /**  
     * Case 3 documentation.  
     *  
     * - Remark 1.  
     *  
     * - Remark 2.  
     */  
    kHAPEnumType_ReallyReallyLongCase,  
  
    /** Case 4 documentation. */  
    kHAPEnumType_ReallyReallyLongCase2  
} HAP_ENUM_END(uint8_t, HAPEnumType);
```

Functions

- Function names: `HAPPascalCase`

Use concise names and avoid abbreviations. Length does not matter here. If targeting an object, first mention the object, e.g., `HAPAccessoryServerStartBridge` targets a `HAPAccessoryServer` object. In that case, also make sure that the target object is the very first parameter.

- Parameter names: `camelCase`

Maximum parameter length name is 20. Otherwise it will break the table in the documentation.

Use `HAP_UNUSED`, if the argument is not used in the function body. Do not cast to `(void)` for compiler warnings.

Pointers are assumed to always be non-NULL. Use `_Nullable` on pointer arguments that may be NULL.

- Parameter types use fixed size C99 integers, e.g., `uint8_t`, `int16_t`, `size_t`. Avoid `int`, `short`, `unsigned char`. For booleans, use `bool`. Yes, it's just a typedef to `int` in C, but it improves readability.
- Return type must be `HAPError` if the function is expected to fail. If the function returns a value (non-void),

it must be marked with `HAP_RESULT_USE_CHECK`. If the function cannot fail and does not provide any outputs, return void. If unsure whether a function may return an error, start with void return first. Errors that arise from API misuse (invalid arguments, invalid state) crash / assert instead of returning error.

- Every parameter and return value is documented.
 - If a parameter is an out-parameter, `[out]` is attached after the `@param`.
 - If a parameter is an in/out parameter, `[in, out]` is attached after the `@param`.
 - `true` / `false` return values are written in lowercase in documentation.
 - `HAPError` return values are sorted in documentation in the same way as listed in the `HAPError` enum.
- GCC attributes one per line, sorted alphabetically. Apply GCC attributes to both declaration and definition of the function.
- Function bodies are streamlined:
 - Precondition checks: One precondition per line, checked in the order they appear in the function signature. Casts from Refs to internal types should be made as close as possible to where their preconditions are checked. Early returns may also be included as part of the precondition checks block if it makes sense.
 - `HAPError err;`
 - Function body.
- Use early returns if possible.

If returning because of an error, use a `HAPLog` statement to print the reason of the error before throwing. When just re-throwing an error, do not log. The error is already logged when it was thrown. If a post-condition must be checked even in error cases, wrap the function body in a second function that uses early returns, and check the post-condition in the wrapping function.

```
/**  
 * Brief documentation about the function.  
 *  
 * - Remark 1 about special considerations.  
 *  
 * - Remark 2 about special considerations.  
 *  
 * @param arg1 Argument 1 documentation.  
 * @param arg2 Argument 2 documentation.  
 * @param[out] bytes Buffer that will be filled with data.  
 * @param maxBytes Maximum number of bytes that may be filled into the buffer.  
 * @param[in,out] numBytes Effective number of bytes written to the buffer.  
 *  
 * @return kHAPError_None If successful.  
 * @return kHAPError_OutOfResources If the supplied buffer was not large enough.  
 */  
HAP_RESULT_USE_CHECK  
static HAPError HAPDoSomething()  
{  
    size_t arg1,  
    bool arg2 HAP_UNUSED,  
    void *bytes, // _Nullable if bytes are _Nullable  
    size_t maxBytes,  
    size_t *numBytes) // _Nullable if bytes are _Nullable  
{  
    HAPPrecondition(arg1 > 42);  
    HAPPrecondition(bytes);  
  
    HAPError err;  
  
    // ...  
  
    return kHAPError_None;  
}
```

Header Files

- Name: `HAPPascalCase.h`. The file name does not have to be a prefix of all functions.
- If several items form a certain subgroup of a header, consider using a separate header file.
Name: `HAPBaseHeader+CategoryName.h`
- Include header files from `HAP.h`. Include category headers from their corresponding base header.

- Header guards: `HAP_UNDERSCORED_FILE_NAME_H`. Do not forget to update the header guard after file renames. Do not forget to update the log category / subsystem in source files that implement the header after file renames.
- Always include `extern "C"` declaration, even in internal headers. Headers may move around over time and become visible to C++. Also include the declaration when it is not necessary, for consistency. It doesn't hurt.
- Header files are streamlined:
 - Copyright notice. Make sure it is EXACTLY the same in all files, so that automated re-copyrighting scripts work.
 - Empty line.
 - Header guard.
 - Empty line.
 - `extern "C"` declaration.
 - `#include <system_header.h>`, sorted alphabetically. If some are in a subdirectory, place them in the end.
System headers must only be included in `HAPBase.h` and in platform-specific code!
 - Empty line, if system headers were included.
 - `#include "platform_header.h"`, sorted alphabetically. This is for headers coming from external dependencies.
 - Empty line, if platform headers were included.
 - `#include "HAP.h" / #include "HAPPlatform.h"`, and categories of own header files, sorted alphabetically.
 - Empty line.
 - Enter `assume_nonnull` block.
 - Header file contents.
 - Exit `assume_nonnull` block.
 - Empty line.
 - If necessary, further `#include` statements, followed by empty line.
 - Complete `extern "C"` declaration.
 - Empty line.
 - Complete header guard.
 - Empty line.

```

#ifndef HAP_HEADER_FILE_NAME_H
#define HAP_HEADER_FILE_NAME_H

#ifdef __cplusplus
extern "C" {
#endif

#include <system_header.h>

#include "platform_header.h"

#include "HAPPlatform.h"

#if __has_feature(nullability)
#pragma clang assume_nonnull begin
#endif

// Header file contents.

#if __has_feature(nullability)
#pragma clang assume_nonnull end
#endif

#ifdef __cplusplus
}
#endif

#endif

```

Switch Statements

- When switching over a closed enum, do not add a default case. Try to return from every case, and place a fatal error after the switch. This ensures compiler warnings when new enum cases are added but are not explicitly handled.
- Use a separate scope for every case. (Empty cases are allowed to be condensed together, e.g., case 'c' / 'd' below).

- After the closing bracket of each case's scope, put either a `break;` statement or a `// Fallthrough.` comment.
- When switching on enumerations, add a default case like `default: HAPFatalError();` or `default: false` as the final line to cover unexpected values.

```
switch (foo) {
    case 'a': {
        // Handle case.
        break;
    }
    case 'b': {
        // Handle case.
        break;
    }
    case 'c':
    case 'd': {
        // Handle cases.
    } // Fallthrough.
    case 'e': {
        // Handle case.
        break;
    }
}
```

- Exceptions are simple switch statements with early returns where putting an extra break statement after the case's scope does not help with readability. If there are a lot of cases and the switch is just a conversion table from one constant to another it can also be considered to remove the brackets to avoid spanning the switch across pages.

```
static const char *GetCurrentHeatingCoolingStateDescription(
    HAPCharacteristicValue_CurrentHeatingCoolingState state)
{
    switch (state) {
        case kHAPCharacteristicValue_CurrentHeatingCoolingState_Off:
            return "Off.";
        case kHAPCharacteristicValue_CurrentHeatingCoolingState_Heat:
            return "Heat. The Heater is currently on.";
        case kHAPCharacteristicValue_CurrentHeatingCoolingState_Cool:
            return "Cool. Cooler is currently on.";
        default: HAPFatalError();
    }
}
```

Goto Statements

- In general, goto statements should be avoided. Especially, goto statements must not be used to jump backwards. However, in the following scenarios, goto statements may be used.
- If a postcondition needs to be checked or final cleanup needs to be performed before returning from a function, instead of using an early return, a goto statement may be used to handle that finalization.
- Goto statements may be used to break out of nested loops.
- Goto statements may also be used to break from a case statement of a switch on an enumeration. This allows placing a `HAPFatalError()` immediately after the switch without providing a default case. Not providing a default case allows the compiler to emit warnings if an enumeration value is not handled. Placing a `HAPFatalError()` and goto label immediately after the switch allows catching unexpected value.
- Goto statements may be used when multiple cases of a switch prepare some variables but otherwise share their implementation. In this case the goto label shall be placed immediately after the cases sharing the code, before the definition of the next case label.
- Goto labels should use camelCase.

Magic numbers

- Number literals are avoided in favor of named constants, except where a literal is more transparent, e.g.:
 - obvious offsets like `+/- 1`
 - offsets or shift distances and masks for bit manipulation like this: `mantissa = (x >> 9) & 0xFFFF;`

- POSIX success (0) and most important error (-1) numbers
- Test code is allowed to contain magic numbers.

Dynamic memory allocation

- There are only a few places in the PAL where we do dynamic memory allocation, none in the HAP Library.
- Use the macro `HAPPlatformFreeSafe` to deallocate memory in cases where memory is allocated dynamically with libc.
- Use similar macros for third-party library deallocation.

Use of parameters

- All public functions are documented with Doxygen comments.
- All const pointers and non-pointer values are inputs (caller provides data).
- All non-const pointers are outputs or inouts.
- In Doxygen, parameters where the caller must supply input data are not annotated.
- In Doxygen, parameters where the caller should use output data are annotated with [out].
- In Doxygen, parameters where the caller must supply input data and should use output data are annotated with [in,out], except for "this" pointers which are the first parameters of functions that work with object-like data types.

Use of void parameters

- We normally use typed parameters.
- If byte buffers are returned, void is used instead, to not force clients to use type casts. This is the same style as Apple's mDNSResponder.

Opaque structures

- Internal structures that are exposed publicly should be hidden by following the opaque pattern.
- Public header: Use the `HAP_OPAQUE` define with the first arguments defining the size in multiples of 8. The associated data is aligned on 8 bytes. Name ends in `Ref`.

```
typedef HAP_OPAQUE(24) HAPFooRef;
```

- Private header: Specify real structure without "Ref" in name and static assert that it fits into a Ref.

```
typedef struct { ... } HAPFoo;
```

```
HAP_STATIC_ASSERT(sizeof (HAPFooRef) >= sizeof (HAPFoo), HAPFoo);
```

- Functions: Always pass around refs. If internal fields need to be accessed, create second variable with `_` postfix.

```
HAPFooRef *foo_;  
HAPFoo *foo = (HAPFoo *) foo_;
```

Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies within all project spaces, and it also applies when an individual is representing the project or its community in public spaces.

Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the open source team at opensource-conduct@group.apple.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the [Contributor Covenant](#), version 1.4, available at <https://www.contributor-covenant.org/version/1/4/code-of-conduct.html>

The following organizations and individuals have contributed to the HomeKit ADK:

- Oberon microsystems AG
- Apple Inc.

By submitting a pull request, you represent that you have the right to license your contribution to Apple and the community, and agree by submitting the patch that your contributions are licensed under the [Apache 2.0 license](#).

Before submitting the pull request, please make sure you have tested your changes and that they follow the project [guidelines for contributing code](#).

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally

submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution." "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or

documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions.

Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability

incurred by, or claims asserted against, such Contributor by reason
of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and limitations under the License.

HomeKit Accessory Development Kit (ADK)

The HomeKit ADK is used by silicon vendors and accessory manufacturers to build HomeKit compatible devices.

The HomeKit ADK implements key components of the HomeKit Accessory Protocol (HAP), which embodies the core principles Apple brings to smart home technology: security, privacy, and reliability.

The HomeKit Open Source ADK is an open-source version of the HomeKit Accessory Development Kit. It can be used by any developer to prototype non-commercial smart home accessories. For commercial accessories, accessory developers must continue to use the commercial version of the HomeKit ADK available through the MFi Program.

Go to the [Apple Developer Site](#) if you like to learn more about developing HomeKit-enabled accessories and apps.

Documentation

- Please go through [Developing with ADK](#) before starting development with HomeKit ADK
- [Platform Abstraction Layer](#)

Darwin PAL

Compile

`make all`

Run

Linux PAL

Compile

```
make TARGET=Linux apps
```

Raspberry Pi

Prerequisites

Run this to create the SD card image (Linux+patches) and a Docker container (your build environment).

Make sure that Docker is running before running this setup or it will fail and you will have to start over!

This will take about an hour and will stop for input many times. It will also require you to swap the card out twice.

```
# Run the Docker app. It is required for docker import.  
./Tools/raspi_sdcard_setup.sh
```

Compile

```
make TARGET=Raspi all
```

If docker doesn't find "dev-test/raspividk-base", run the sdcard setup and make sure to do the docker import at the end of the script.

Install

After building, run this to install the build products to your RaspPi. (-n is for the hostname of your RaspPi and -p is the SSH password; both of these were chosen during the initial raspi_sdcard_setup.sh install).

```
./Tools/install.sh \  
-d raspi \  
-a Output/Raspi-armv6k-unknown-linux-gnueabihf/Debug/IP/Applications/Lightbulb.OpenSSL \  
-n raspberrypi \  
-p pi
```

Make options

Command	Description	Default
make <target>	<ul style="list-style-type: none">apps - Build all appstest - Build unit testsall - Build apps and unit tests	all
make APPS=<application>	Space delimited names of the apps to compile. Example: make APPS="Lightbulb Lock"	All applications
make BUILD_TYPE=<build_type>	Build type: <ul style="list-style-type: none">DebugTestRelease	Debug
make LOG_LEVEL=<level>	<ul style="list-style-type: none">0 - No logs are displayed1 - Error and Fault-level logs are displayed2 - Error, Fault-level and Info logs are displayed3 - Error, Fault-level, Info and Debug logs are displayed	<ul style="list-style-type: none">3 - For debug build1 - For test build0 - For release build

make PROTOCOLS=<protocol>	<p>Space delimited protocols supported by the applications:</p> <ul style="list-style-type: none"> • BLE • IP <p>Example: make PROTOCOLS="IP BLE"</p>	All protocols
make TARGET=<platform>	<p>Build for a given target platform:</p> <ul style="list-style-type: none"> • Darwin • Linux • Raspi 	Build for the host Platform
make USE_HW_AUTH=<enable>	<p>Build with hardware authentication enabled:</p> <ul style="list-style-type: none"> • 0 - Disable • 1 - Enable 	Disabled
make USE_NFC=<enable>	<p>Build with NFC enabled:</p> <ul style="list-style-type: none"> • 0 - Disable • 1 - Enable 	Disabled

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#)

the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.

3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

INA3221 I2C Driver

A Mongoose library for Texas Instruments' 3-channel current sensor.

Implementation details

The `INA3221` is a simple I2C device that measures current through shunt resistors on three separate channels. It is able to measure the voltage on each channel (max 26V, in 8mV increments) and the voltage drop over the shunt resistor in 40uV increments with a range of 163.8mV, and is quite precise.

Using a 0.1Ohm shunt resistor, the maximum current that can be measured is 1.638A. Each channel can be enabled/disabled and polled individually.

API Description

Create an `INA3221` object using `mgos_ina3221_create()`, set the shunt resistor values for each channel using `mgos_ina3221_set_shunt_resistance()`, then call `mgos_ina3221_get_bus_voltage()` to get the bus voltage, call `mgos_ina3221_get_shunt_voltage()` to get the voltage drop over the shunt resistor, and `mgos_ina3221_get_current()` to get the current flowing through the channel.

Valid channels are 1, 2 or 3 (ie. they are 1-based).

Example application

```

#include "mgos.h"
#include "mgos_config.h"
#include "mgos_ina3221.h"

static void ina3221_timer_cb(void *user_data) {
    struct mgos_ina3221 *sensor = (struct mgos_ina3221 *)user_data;

    if (!sensor) return;

    for (uint8_t i=1; i<4; i++) {
        float bus, shunt, current, res;
        mgos_ina3221_get_bus_voltage(sensor, i, &bus);
        mgos_ina3221_get_shunt_resistance(sensor, i, &res);
        mgos_ina3221_get_shunt_voltage(sensor, i, &shunt);
        mgos_ina3221_get_current(sensor, i, &current);
        LOG(LL_INFO, ("Chan[%u]: Vbus=% .3f V Vshunt=% .0f uV Rshunt=% .3f Ohm Ishunt=% .1f mA",
                     i, bus, shunt*1e6, res, current*1e3));
    }
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_ina3221 *sensor;

    sensor = mgos_ina3221_create(mgos_i2c_get_global(), mgos_sys_config_get_ina3221_i2caddr());
    if (!sensor) {
        LOG(LL_ERROR, ("Could not create INA3221 sensor"));
        return false;
    }

    mgos_set_timer(1000, true, ina3221_timer_cb, sensor);

    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Debug over Bluetooth GATT Service

Overview

This library provides various debug functions over Generic Attribute Service (GATT) Bluetooth Low-Energy (BLE) service.

The service is designed to be usable with any generic BLE mobile app that supports GATT, e.g. BLE Scanner ([Android](#), [iOS](#)).

Note: Default BT configuration is permissive. See <https://github.com/mongoose-os-libs/bt-common#security> for a better idea.

Attribute description

The service UUID is `5f6d4f53-5f44-4247-5f53-56435f49445f`, which is a representation of a 16-byte string `_mOS_DBG_SVC_ID_`.

At present, only one characteristic is defined:

- `306d4f53-5f44-4247-5f6c-6f675f5f5f30` (`0mOS_DBG_log__0`) - a read/notify attribute that returns last debug log record when read. It also sends notifications with log messages as they are printed.
 - *Note:* Reading large messages is supported, but for notifications to be useful you will most likely want to set higher MTU.

File-logger

File-logger is a library which implements log file rotation: it maintains max X log files of max size Y, so that you always have latest logs from the device persisted on the filesystem. By default there are max 10 files, prefixed with `log_`, each of max size 4000 bytes.

See [mos.yml](#) for the possible options. At least you'd have to enable this lib in your app's `mos.yml`, like this:

Note that for the message to get to file logger it needs to be allowed by `debug.level` and `debug.event_level`. File logger then examines `file_logger.level` and `file_logger.include` to make final determination.

`file_logger.include` is a list of substrings to match, so a value of `mg_rpc.c,Tick,Tock` will match lines containing any of "Tick", "Tock" or "mg_rpc.c".

```
libs:
  - origin: https://github.com/mongoose-os-libs/file-logger

config_schema:
  - ["file_logger.enable", true]
```

Fetching logs

See [this script](#). libbpread

libbpread is a tiny, fast bitmap (.bmp) image file loader, written from scratch in portable C (see below), with no dependencies. Its default behavior is compatible with OpenGL texture functions, making it ideal for use in simple games. It handles any valid bit depth (1, 4, 8, 16, 24, or 32), and can even load the alpha channel from 16- and 32-bit bitmaps. There is no support for RLE or other compression schemes yet.

<https://github.com/chazomaticus/libbpread>

Documentation

To use, simply copy `bpread.c` and `bpread.h` into your project and add them to the build.

The API exposed by `bpread.h` is described below. The same description can be found in the header itself.

bpread()

Loads the specified bitmap file from disk and fills out a `bpread_t` struct with data about it.

```
int bpread(const char * bmp_file,
           unsigned int flags,
           bpread_t * p_bmp_out);
```

- `bmp_file`: The filename of the bitmap file to load.

- `flags`: Any `BMPREAD_*` flags, combined with bitwise OR. Specify 0 (or `BMPREAD_ALPHA` if you want an alpha channel) for standard, OpenGL compliant behavior.
- `p_bmp_out`: Pointer to a `bmpread_t` struct to fill with information. Its contents on input are ignored. Must be freed with `bmpread_free()` when no longer needed.

Returns 0 if there's an error (file doesn't exist or is invalid, i/o error, etc.), or nonzero if the file loaded ok.

The file must be a Windows 3 (not NT) or higher format bitmap file with any valid bit depth (1, 4, 8, 16, 24, or 32), and must not be compressed (no RLE).

Default behavior is for `bmpread()` to return `data` in a format directly usable by OpenGL texture functions, e.g. `glTexImage2D`, format `GL_RGB` (or `GL_RGBA` if `BMPREAD_ALPHA` is in `flags`), type `GL_UNSIGNED_BYTE`. This implies a few oddities:

- Lines are ordered bottom-first. To return data starting with the top line like you might otherwise expect, pass `BMPREAD_TOP_DOWN` in `flags`.
- Lines are padded to span a multiple of four bytes. To return data with no padding, pass `BMPREAD_BYTE_ALIGN` in `flags`.
- Images with a width or height that isn't a power of 2 will fail to load. To allow loading images of any size, pass `BMPREAD_ANY_SIZE` in `flags`.

Note that passing any of these flags may cause the output to be unusable as an OpenGL texture, which may or may not matter to you.

Most bitmap files can't include an alpha channel, so the default behavior is to ignore any alpha values present in the file. Pass `BMPREAD_ALPHA` in `flags` to capture alpha values from the file; in case of an absent alpha channel, alpha values are output as 255 (this can be changed by redefining `BMPREAD_DEFAULT_ALPHA` in `bmpread.c`). This allows fully loading 16- and 32-bit bitmaps, which *can* include an alpha channel.

bmpread_free()

Frees memory allocated during `bmpread()`. Call `bmpread_free()` when you are done using the `bmpread_t` struct (e.g. after you have passed the data on to OpenGL).

```
void bmpread_free(bmpread_t * p_bmp);
```

- `p_bmp`: The pointer you previously passed to `bmpread()`.

bmpread_t

The struct filled by `bmpread()`. Holds information about the image's pixels.

```
typedef struct bmpread_t
{
    int width;
    int height;

    unsigned int flags;

    unsigned char * data;

} bmpread_t;
```

- `width`: Width in pixels.
- `height`: Height in pixels.
- `flags`: `BMPREAD_*` flags, combined with bitwise OR, that affect the format of data. These are set to the flags passed to `bmpread()`.
- `data`: A buffer holding the pixel data of the image.

By default, each pixel spans three bytes: the red, green, and blue color components in that order. However, with `BMPREAD_ALPHA` set in `flags`, each pixel spans four bytes: the red, green, blue, and alpha components in that order.

Pixels are ordered left to right sequentially. By default, the bottom line comes first, proceeding upward. However, with `BMPREAD_TOP_DOWN` set in `flags`, the top line comes first, proceeding downward instead.

Lines by default must span a multiple of four bytes. If the image width and pixel span don't yield a multiple of four (a non-issue for `BMPREAD_ALPHA` with four bytes per pixel), the end of each line is padded with up to three unused bytes to meet the requirement. For example, each line of an image three pixels wide, loaded without `BMPREAD_ALPHA`, will span 12 bytes (3 pixels * 3 (RGB) channels per pixel = 9, padded with 3 bytes up to the next multiple of 4). However, this behavior is disabled with `BMPREAD_BYTE_ALIGN` set in `flags`, in which case all lines span exactly `width * pixel_span` bytes.

Flags

Flags for `bmpread()` and `bmpread_t`. Combine with bitwise OR.

- `BMPREAD_TOP_DOWN`: Output data as top line first (default is bottom line first).

```
#define BMPREAD_TOP_DOWN 1u
```

- `BMPREAD_BYTE_ALIGN`: Don't pad lines to span a multiple of four bytes (default does pad).

```
#define BMPREAD_BYTE_ALIGN 2u
```

- `BMPREAD_ANY_SIZE`: Allow loading of any size bitmap (default is bitmaps must be $2^n \times 2^m$).

```
#define BMPREAD_ANY_SIZE 4u
```

- `BMPREAD_ALPHA`: Load and output an alpha channel (default is just color channels).

```
#define BMPREAD_ALPHA 8u
```

Example

Here's a code snippet showing how libbmpread might be used to create an OpenGL texture from a bitmap file on disk:

```

#include <stdio.h>
#include <GL/gl.h>
#include "bmpread.h"

/* Load the specified bitmap file from disk and copy it into an OpenGL texture.
 * Return the GLuint representing the texture.
 */
GLuint LoadTexture(const char * bitmap_file)
{
    GLuint texture = 0;
    bmpread_t bitmap;

    if(!bmpread(bitmap_file, 0, &bitmap))
    {
        fprintf(stderr, "%s: error loading bitmap file\n", bitmap_file);
        exit(1);
    }

    /* At this point, bitmap.width and .height hold the pixel dimensions of the
     * file, and bitmap.data holds the raw pixel data in RGB triplets.
     */

    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    glTexImage2D(GL_TEXTURE_2D, 0, 3, bitmap.width, bitmap.height, 0,
                 GL_RGB, GL_UNSIGNED_BYTE, bitmap.data);

    bmpread_free(&bitmap);

    return texture;
}

void SomeInitFunction(void)
{
    GLuint tex1 = LoadTexture("texture1.bmp");
    // ...
}

```

See also the `example` directory for a full, compiling example that displays a bitmap named on the command line on an OpenGL quad using GLUT. To build and run:

```

cd example
make
./bmpread-example <bmpfile>

```

Some example bitmap files are included there as well.

Portability/Security Notes

`libbmpread` is written to be maximally ANSI C (C89/C90) portable, minimizing undefined and implementation-defined behavior. It's also written to be able to be compiled as C++, so it can be used in C++ projects with a minimum of fuss. It should work in most environments--I believe there are only two assumptions the code makes that aren't guaranteed by the C standard: 1) `CHAR_BIT == 8` (note: if this is violated, compilation will fail), and 2) two's complement integer storage; deviations from these assumptions are extremely rare in the wild. I've also made use of `<stdint.h>` and `ptrdiff_t`, which I believe are technically C99 features, but are common in practice even for non-compliant compilers.

I've taken every precaution to prevent common bugs that can have security impact, such as integer overflows that might lead to buffer overruns. I believe it's impossible to cause libbpread to do anything besides properly load a file or fail with error, even on maliciously crafted files. I haven't done a thorough audit, nor am I an expert at writing hardened "C/C++" code, so take my belief with a grain of salt.

Tests

To run the test suite:

```
cd test  
make
```

The last line will read All tests passed! if everything was ok. Be patient: I test an unoptimized function over its entire 32-bit numeric range, so it can take a minute or two to finish.

There are some basic unit tests, but more work is needed to test the behavior of the public API. The tests are compiled as both portable C and portable C++ code to cover compilation in each, and each resulting test binary is run to check its runtime behavior.

Authors

- [Charles Lindsay](#) (maintainer)
- [xFrednet](#) (added support for 16- and 32-bit files)

Enjoy!

Wrapper lib for Mongoose-OS to load BMP images

This is a Mongoose OS library to use "[libbpread](#)" by Charles Lindsay "chazomaticus" seamlessly in Mongoose OS applications.

The third-party software is integrated without modification under the directory `third_party`. The documentation for the underlying lib can also be found in the `third_party` directory. The external library is 100% original and not changed (will be included as submodule in a future version), so it's easier to integrate updates later, if they show up.

Be aware: this library might be subject of change, so use it at your own risk!

Dependencies

This library has a dependency:

- [libbpread](#)

Documentation

These functions are available via C calls:

```
// create the needed structure to hold the data:  
bmpread_t* mgos_bmp_loader_create(void)  
  
// load an BMP image  
bool mgos_bmp_loader_load(bmpread_t* p_bmp, const char* bmp_file, unsigned int flags)  
  
// free the allocated space  
void mgos_bmp_loader_free(bmpread_t* p_bmp)  
  
// get the plain data pointer of the image  
uint8_t* mgos_bmp_loader_get_data(bmpread_t* p_bmp)  
  
// get the width and height of the image (wrappers for FFI)  
uint32_t mgos_bmp_loader_get_width(bmpread_t* p_bmp)  
uint32_t mgos_bmp_loader_get_height(bmpread_t* p_bmp)  
  
// rotate the image in 90° steps  
bool mgos_bmp_loader_rotate(bmpread_t* p_bmp, uint16_t angle)
```

A MJS API is included as well:

Include the API with

```
load('api_bmp.js');
```

in your own MJS file like `init.js`.

These functions are available as MJS API calls:

```
// the object containing the API
let BMP_LOADER = { ... };

// create a new object with the needed structure
create: function () { ... }

// load a BMP file
load: function (file, flags) { ... }

// free the underlying structure and the bitmap data
free: function () { ... }

// get a pointer to the raw bitmap data
getData: function () { ... }

// get a certain pixel of the bitmap as pointer to the bytes
getPixel: function (x, y) { ... }

// get the width of the image
getWidth: function () { ... }

// get the height of the image
getHeight: function () {...}
```# Inventek es-WiFi module driver for Mongoose OS
```

Driver for the [Inventek eS-WiFi ISM43xxx module series](<https://www.inventeksys.com/es-wifi-support/>) (ISM4319-M3-xxx, ISM

Only SPI interface is supported for now.

```
Configuration
<p class="mume-header" id="configuration"></p>
```

In addition to the common wifi STA and AP options, this library adds a number of its own settings.

You need to configure at least CS and DRDY pins, others are optional.

See [manifest](mos.yml) for definitions and examples.

```
SNTP library for Mongoose OS
<p class="mume-header" id="sntp-library-for-mongoose-os"></p>
```

When included to a Mongoose OS application, this library fetches the current time from the SNTP server (by default, `time.google.com` is used) every time the Internet connection is established, and adjusts the device time.

See `mos.yml` for the SNTP configuration available.

```
Board definitions library
<p class="mume-header" id="board-definitions-library"></p>
```

This library contains configuration tweaks for different boards supported by Mongoose OS.

```
Python CryptoAuthLib Module Testing
<p class="mume-header" id="python-cryptauthlib-module-testing"></p>
```

```
Introduction
<p class="mume-header" id="introduction"></p>
```

These tests are designed to only test the python interface to the library and are not designed to test the library itself which is covered by the main cryptauthlib tests

```
Running
<p class="mume-header" id="running"></p>
```

The best way to run the test suite is to use [tox](<https://tox.readthedocs.io/en/latest/>) which can be easily installed with pip:

```
$ pip install tox
```

From the python folder:

```
:~/cryptoauthlib/python $ tox
```

It is possible to directly run tests but requires more setup

1) Install pytest

```
$ pip install pytest
```

2) Modify the PYTHONPATH environment variable

Windows:

```
cryptoauthlib/python> set PYTHONPATH=<path_to>/cryptoauthlib/python
```

Linux:

```
export PYTHONPATH ={PYTHONPATH}:<path_to>/cryptoauthlib/python
```

3) Run the tests

```
$ pytest -vv
```

### Test options

<p class="mume-header " id="test-options"></p>

There are additional options that can be invoked with the tests that define what tests will be run

1) --with-lib will attempt to run tests against the compiled c library.

These tests are good for detecting possible platform incompatibilities between the C compiler and the expectations of python

2) --with-device will attempt to invoke some tests with a real attached device

These tests are restricted to only the minimum required to verify the python to library connectivity and are only meant to detect situations that can not be determined from the library tests alone.

# Python CryptoAuthLib module

<p class="mume-header " id="python-cryptoauthlib-module"></p>

## Introduction

<p class="mume-header " id="introduction-1"></p>

This module provides a thin python ctypes layer to evaluate the cryptoauthlib interface to Microchip CryptoAuthentication devices.

### Code Examples

<p class="mume-header " id="code-examples"></p>

Code examples for python are available on github as part of

[CryptoAuthTools](<https://github.com/MicrochipTech/cryptoauthtools>) under the [python/examples](<https://github.com/MicrochipTech/cryptoauthtools/tree/master/python/examples>) directory

## Installation

<p class="mume-header " id="installation-1"></p>

### CryptoAuthLib python module can be installed through Python's pip tool:

<p class="mume-header " id="cryptoauthlib-python-module-can-be-installed-through-pythons-pip-tool"></p>

```
pip install cryptoauthlib
```

### To upgrade your installation when new releases are made:

<p class="mume-header " id="to-upgrade-your-installation-when-new-releases-are-made"></p>

```
pip install -U cryptoauthlib
```

### If you ever need to remove your installation:

<p class="mume-header " id="if-you-ever-need-to-remove-your-installation"></p>

```
pip uninstall cryptoauthlib
```

```
What does python CryptoAuthLib package do?
```

```
<p class="mume-header " id="what-does-python-cryptoauthlib-package-do"></p>
```

CryptoAuthLib module gives access to most functions available as part of standard cryptoauthlib (which is written in 'C'). These python functions for the most part are very similar to 'C' functions. The module in short acts as a wrapper over the 'C' cryptoauth library functions.

Microchip cryptoauthlib product page:

[Link](<http://www.microchip.com/SWLibraryWeb/product.aspx?product=CryptoAuthLib>)

```
Supported hardware
```

```
<p class="mume-header " id="supported-hardware"></p>
```

- [AT88CK101](<http://www.microchip.com/DevelopmentTools/ProductDetails/AT88CK101SK-MAH-XPRO>)

- [CryptoAuthentication SOIC XPRO Starter Kit (DM320109)](<https://www.microchip.com/developmenttools/ProductDetails/DM320109>)

```
Supported devices
```

```
<p class="mume-header " id="supported-devices"></p>
```

The family of devices supported currently are:

- [ATSHA204A](<http://www.microchip.com/ATSHA204A>)

- [ATECC108A](<http://www.microchip.com/ATECC108A>)

- [ATECC508A](<http://www.microchip.com/ATECC508A>)

- [ATECC608A](<http://www.microchip.com/ATECC608A>)

```
Using cryptoauthlib python module
```

```
<p class="mume-header " id="using-cryptoauthlib-python-module"></p>
```

The following is a 'C' code made using cryptoauthlib 'C' library.

```
```C
#include "cryptoauthlib.h"

void main()
{
    ATCA_STATUS status;
    uint8_t revision[4];
    uint8_t randomnum[32];

    status = atcab_init(cfg_ateccx08a_kitcdc_default);
    if (status != ATCA_SUCCESS)
    {
        printf("Error");
        exit();
    }

    status = atcab_info(revision);
    if (status != ATCA_SUCCESS)
    {
        printf("Error");
        exit();
    }

    status = atcab_random(randomnum);
    if (status != ATCA_SUCCESS)
    {
        printf("Error");
        exit();
    }
}
```

The same code in python would be:

```

from cryptoauthlib import *

ATCA_SUCCESS = 0x00
revision = bytearray(4)
randomnum = bytearray(32)

# Locate and load the compiled library
load_cryptoauthlib()

assert ATCA_SUCCESS == atcab_init(cfg_ateccx08a_kithid_default())

assert ATCA_SUCCESS == atcab_info(revision)
print(''.join(['%02X ' % x for x in revision]))

assert ATCA_SUCCESS == atcab_random(randomnum)
print(''.join(['%02X ' % x for x in randomnum]))

```

In the above python code, "import cryptoauthlib" imports the python module. `load_cryptoauthlib()` function loads the compiled library. The `load_cryptoauthlib()` is a function that you will not see in the 'C' library, this is a python specific utility function and is required for python scripts to locate and load the compiled library.

In Summary

Step I: Import the module

```
from cryptoauthlib import *
```

Step II: Initialize the module

```
load_cryptoauthlib()

assert ATCA_SUCCESS == atcab_init(cfg_ateccx08a_kithid_default())
```

Step III: Use Cryptoauthlib APIs

Call library APIs of your choice

Code portability

Microchip's CryptoAuthentication products can now be evaluated with the power and flexibility of python. Once the evaluation stage is done the python code can be ported to 'C' code.

As seen above the python API maintains a 1 to 1 equivalence to the 'C' API in order to ease the transition between the two.

Cryptoauthlib module API documentation

help() command

All of the python function's documentation can be viewed through python's built in `help()` function.

For example, to get the documentation of `atcab_info()` function:

```
>>> help(cryptoauthlib.atcab_info)
Help on function atcab_info in module cryptoauthlib.atcab:
```

```
atcab_info(revision)
Used to get the device revision number. (DevRev)
```

Args:
revision 4-byte bytearray receiving the revision number
from the device. (Expects bytearray)

Returns:
Status code

dir() command

The dir command without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object. For example
dir(cryptoauthlib) will return all the methods available in the cryptoauthlib module.

Code Examples

Code examples for python are available on github as part of
[CryptoAuthTools](#) under the
python/examples directory

Tests

Module tests can be located in the [python/tests](#)
of the main cryptoauthlib repository. The [README.md](#)
has details for how to run the tests. The module tests are not comprehensive for the entire functionality
of cryptoauthlib but rather are meant to test the python module code only against the library to ensure
the interfaces are correct and ctypes structures match the platform.

basic directory - Purpose

The purpose of this directory is to contain the files implementing the APIs for
a basic interface to the core CryptoAuthLib library.

High-level functions like these make it very convenient to use the library when
standard configurations and defaults are in play. They are the easiest to use
when developing examples or trying to understand the "flow" of an
authentication operation without getting overwhelmed by the details.

This makes simple jobs easy and if you need more sophistication and power, you
can employ the full power of the CryptoAuthLib object model.

See the Doxygen documentation in cryptoauthlib/docs for details on the API of the Basic commands. crypto directory - Purpose

This directory contains software implementations of cryptographic functions.

The functions at the base level are wrappers that will point to the final implementations of the software crypto functions.

HAL Directory - Purpose

This directory contains all the Hardware Abstraction Layer (HAL) files used to adapt the upper levels of atca-ng and abstractions to physical hardware.

HAL contains physical implementations for I2C, SWI, SPI, UART and timers for specific hardware platforms.

Include just those HAL files you require based on platform type.

CryptoAuthLib Supported HAL Layers

HAL Layers files are combined into groups. Initial group is generic files that are typically included in a project.

Files are then broken out by uController Family and or Operating System Interface.

Protocol Files	Interface	Files	API	Notes
atca		atca_hal.c/h		For all projects
kit protocol		kit_protocol.c/h		For all Kit Protocol projects
		kit_phy.h		
		hal_i2c_bitbang.c/h	ASF	For all I2C Bitbang projects
		hal_swi_bitbang.c/h	ASF	For all SWI Bitbang projects

Most microcontrollers supported by [Atmel START](#)

have generic drivers depending on the interface.

START Micros	Interface	Files	API	Notes
		hal_timer_start.c	START	Timer implementation
	I2C	hal_i2c_start.c/h	START	
	SWI	swi_uart_start.c/h	START	SWI using UART

AVR Micros	Interface	Files	API	Notes
at90usb1287	I2C	hal_at90usb1287_i2c_asf.c/h	ASF	
		hal_at90usb1287_timer_asf.c	ASF	
	SWI	swi_uart_at90usb1287_asf.c/h	ASF	
xmega_a3bu	I2C	hal_xmega_a3bu_i2c_asf.c/h	ASF	
		hal_xmega_a3bu_timer_asf.c	ASF	
	SWI	swi_uart_xmaga_a3bu_asf.c/h	ASF	

SAM Micros	Interface	Files	API	Notes
sam4s	I2C	hal_sam4s_i2c_asf.c/h	ASF	
		hal_sam4s_timer_asf.c	ASF	

samb11	I2C	hal_samb11_i2c_asf.c/h	ASF	
		hal_samb11_timer_asf.c	ASF	
samd21	I2C	hal_samd21_i2c_asf.c/h	ASF	
		hal_samd21_timer_asf.c	ASF	For all samd21 ASF projects
samd21	I2C	i2c_bitbang_samd21.c/h	ASF	For samd21 I2C bitbang projects
samd21	SWI	swi_bitbang_samd21.c/h	ASF	For samd21 SWI bitbang projects
samd21	SWI	swi_uart_samd21.c/h	ASF	For samd21 SWI uart projects
samg55	I2C	hal_samg55_i2c_asf.c/h	ASF	
		hal_samg55_timer_asf.c	ASF	
samv71	I2C	hal_samv71_i2c_asf.c/h	ASF	
		hal_samv71_timer_asf.c	ASF	

PIC Micros	Interface	Files	API	Notes
pic32mx695f512h	I2C	hal_pic32mx695f512h.c/h	plib.h	For pic32mx695f512h Standalone Mplab projects
		hal_pic32mx695f512h_timer.c	plib.h	For pic32mx695f512h Standalone Mplab projects
PIC32MZ2048	I2C	hal_pic32mz2048efm_i2c.c/h		
		hal_pic32mz2048efm_timer.c		

OS	Interface	Files	API	Notes
MS Windows	kit-cdc	hal_win_kit_cdc.c/h	windows.h	For all windows USB CDC projects
MS Windows	kit-hid	hal_win_kit_hid.c/h	windows.h	For all windows USB HID projects
			setupapi.h	
MS Windows		hal_win_timer.c	windows.h	For all windows projects
Linux	I2C	hal_linux_i2c_userspace.c/h	i2c-dev	
Linux	kit-cdc	hal_linux_kit_cdc.c/h	fopen	For USB Linux CDC projects
Linux	kit-hid	hal_linux_kit_hid.c/h	udev	For USB Linux HID Projects
Linux/Mac		hal_linux_timer.c		For all Linux/Mac projects
All	kit-hid	hal_all_platforms_kit_hidapi.c/h	hidapi	Works for Windows, Linux, and Mac
freeRTOS		hal_freertos.c		freeRTOS common routines

Secure boot using ATECC608A

The SecureBoot command is a new feature on the

[ATECC608A](#) device compared

to earlier CryptoAuthentication devices from Microchip. This feature helps the MCU to identify fraudulent code installed on it. When this feature is implemented, the MCU can send a firmware digest and signature to the ATECC608A. The ATECC608A validates this information (ECDSA verify) and responds to host with a yes or no answer.

The ATECC608A provides options to reduce the firmware verification time by storing the signature or digest after a good full verification (FullStore mode of the SecureBoot command).

- When the ATECC608A stores the digest (SecureBootMode is FullDig), the host only needs to send the firmware digest, which is compared to the stored copy. This skips the comparatively lengthy ECDSA verify, speeding up the secure boot process.
- When the ATECC608A stores the signature (SecureBootMode is FullSig), the host only needs to send the firmware digest, which is verified against the stored signature using ECDSA. This saves time by not needing to send the signature in the command over the bus.

The ATECC608A also provides wire protection features for the SecureBoot command, which can be used to encrypt the digest being sent from the host to the ATECC608A and add a MAC to the verify result coming back to the host so it can't be forced to a success state. This feature makes use of a shared secret between the host and ATECC608A, called the IO protection key.

The secure boot feature can be easily integrated to an existing project. The project should include the following files from the secure_boot folder:

- `secure_boot.c`
- `secure_boot.h`
- `secure_boot_memory.h`
- `io_protection_key.h`

The project should also implement the following platform-specific APIs:

- `secure_boot_init_memory()`
- `secure_boot_read_memory()`
- `secure_boot_deinit_memory()`
- `secure_boot_mark_full_copy_completion()`
- `secure_boot_check_full_copy_completion()`
- `io_protection_get_key()`
- `io_protection_set_key()`

The project can set the secure boot configuration with the following defines:

- `SECURE_BOOT_CONFIGURATION`
- `SECURE_BOOT_DIGEST_ENCRYPT_ENABLED`
- `SECURE_BOOT_UPGRADE_SUPPORT`

The secure boot process is performed by initializing CryptoAuthLib and calling the `secure_boot_process()` function.

Implementation Considerations

- Need to perform SHA256 calculations on the host. CryptoAuthLib provides a software implementation in `lib/crypto/atca_crypto_sw_sha2.c`
- When using the wire protection features:
 - The host needs to be able to generate a nonce (number used once). This is the NumIn parameter to the Nonce command that is sent before the SecureBoot command. The ATECC608A can not be used to generate NumIn, but it should come from a good random or non-repeating source in the host.
 - If the host has any protected internal memory, it should be used to store its copy of the IO protection key.

- Secure boot depends on proper protections of the boot loader code in the host. If the code can be easily changed, then the secure boot process can be easily skipped. Boot loader should ideally be stored in an immutable (unchangeable) location like a boot ROM or write-protected flash.
- Note that these APIs don't provision the ATECC608A. They assume the ATECC608A has already been configured and provisioned with the necessary keys for secure boot.

Examples

For more information about secure boot, please see the example implementation project and documentation at:

https://github.com/MicrochipTech/cryptoauth_usecase_securebootapp directory - Purpose

This directory is for application specific implementation of various use cases.

Methods in this directory provide a simple API to perform potentially complex combinations of calls to the main library or API.

IP Protection with Symmetric Authentication

The IP protection can be easily integrated to the existing projects. The user project should include symmetric_authentication.c & symmetric_authentication.h files which contains the api

- **symmetric_authenticate()** - For Performing the authentication between host & device.

User Considerations

- The user should take care on how the master key should be stored on the MCU side.
- The api's in the file doesn't do the provisioning of the chip and user should take care of the provisioning.

With the provisioned cryptoauthentication device and after doing the cryptoauthlib initialisation, user should only be calling the function symmetric_authenticate() with its necessary parameters for the authentication. The returned authentication status should be used in the application.

Examples

For more information about IP protection and its example project refer [Microchip github](https://github.com/MicrochipTech/cryptoauth_usecase_ip_protection)

TNG Functions

This folder has a number of convenience functions for working with TNG devices (currently ATECC608A-MAHTN-T).

These devices have standard certificates that can be

easily read using the functions in tng_atcacert_client.hCryptoAuthLib - Microchip CryptoAuthentication Library

Introduction

This code base implements an object-oriented C library which supports Microchip CryptoAuth devices. The family of devices supported currently are:

- [ATSHA204A](#)
- [ATECC108A](#)
- [ATECC508A](#)
- [ATECC608A](#)

Online documentation is at <https://microchiptech.github.io/cryptoauthlib/>

Latest software and examples can be found at:

- <http://www.microchip.com/SWLibraryWeb/product.aspx?product=CryptoAuthLib>
- <https://github.com/MicrochipTech/cryptoauthtools>

Prerequisite skills:

- strong C programming and code reading
- Atmel Studio familiarity
- Knowledge of flashing microcontrollers with new code
- Familiarity with Microchip CryptoAuth device functionality

Prerequisite hardware to run CryptoAuthLib examples:

- [ATSAMR21 Xplained Pro](#)
or [ATSAMD21 Xplained Pro](#)
- [CryptoAuth Xplained Pro Extension](#)
or [CryptoAuthentication SOIC Socket Board](#)
to accept SOIC parts

For most development, using socketed top-boards is preferable until your configuration is well tested, then you can commit it to a CryptoAuth Xplained Pro Extension, for example. Keep in mind that once you lock a device, it will not be changeable.

There are two major compiler defines that affect the operation of the library.

- ATCA_NO_POLL can be used to revert to a non-polling mechanism for device responses. Normally responses are polled for after sending a command, giving quicker response times. However, if ATCA_NO_POLL is defined, then the library will simply delay the max execution time of a command before reading the response.
- ATCA_NO_HEAP can be used to remove the use of malloc/free from the main library. This can be helpful for smaller MCUs that don't have a heap implemented. If just using the basic API, then there shouldn't be any code changes required. The lower-level API will no longer use the new/delete functions and the init/release functions should be used directly.

Examples

- Watch [CryptoAuthLib Documents](#) for new examples coming online.
- Node Authentication Example Using Asymmetric PKI is a complete, all-in-one example demonstrating all the stages of crypto authentication starting from provisioning the Crypto Authentication device ATECC608A/ATECC508A with keys and certificates to demonstrating an authentication sequence using asymmetric techniques.
<http://www.microchip.com/SWLibraryWeb/product.aspx?product=CryptoAuthLib>

Release notes

08/30/2019

- Added big-endian architecture support
- Fixes to atcah_gen_dig() and atcah_nonce()

05/17/2019

- Added support for TNG devices (cert transforms, new API)
- atcab_write_pub_key() now works when the data zone is unlocked

03/04/2019

- mbed TLS wrapper added
- Minor bug fixes

01/25/2019

- Python JWT support
- Python configuration structures added
- Restructure of secure boot app

01/04/2019

- Added GCM functions
- Split AES modes into separate files
- Bug fix in SWI START driver

10/25/2018

- Added basic certificate functions to the python wrapper.
- Added Espressif ESP32 I2C driver.
- Made generic Atmel START drivers to support most MCUs in START.
- Added AES-CTR mode functions.
- Python wrapper functions now return single values with AtcaReference.
- Added mutex support to HAL and better support for freeRTOS.

08/17/2018

- Better support for multiple kit protocol devices

07/25/2018

- Clean up python wrapper

07/18/2018

- Added ATCA_NO_HEAP define to remove use of malloc/free.
- Moved PEM functions to their own file in atcacert.

- Added wake retry to accomodate power on self test delay.
- Added ca_cert_def member to atcacert_def_s so cert chains can be traversed as a linked list.

03/29/2018

- Added support for response polling by default, which will make commands return faster (define ATCA_NO_POLL to use old delay method).
- Removed atcats related files as they were of limited value.
- Test framework generates a prompt before locking test configuration.
- Test framework puts device to sleep between tests.
- Fixed mode parameter issue in atcah_gen_key_msg().
- ATECC608A health test error code added.

01/15/2018

- Added AES-128 CBC implementation using AES command
- Added AES-128 CMAC implementation using AES command

11/22/2017

- Added support for FLEXCOM6 on SAMG55 driver

11/17/2017

- Added library support for the ATECC608A device
- Added support for Counter command
- atca_basic functions and tests now split into multiple files based on command
- Added support for multiple base64 encoding rules
- Added support for JSON Web Tokens (jwt)
- Fixed atcab_write_enc() function to encrypt the data even when the device is unlocked
- Fixed atcab_base64encode_() for the extra newline
- Updated atcab_ecdh_enc() to work more consistently

07/01/2017

- Removed assumption of SN[0:1]=0123, SN[8]=EE. SN now needs to be passed in for functions in atca_host and atca_basic functions will now read the config zone for the SN if needed.
- Renamed atcab_gendig_host() to atcab_gendig() since it's not a host function. Removed original atcab_gendig(), which had limited scope.
- Fixed atcah_hmac() for host side HMAC calculations. Added atcab_hmac().
- Removed unnecessary ATCDeviceType parameters from some atca_basic functions.
- Added atcacert_create_csr() to create a signed CSR.
- New HAL implementation for Kit protocol over HID on Linux. Please see the Incorporating CryptoAuthLib in a Linux project using USB HID devices section in this file for more information.
- Added atcacert_write_cert() for writing certificates to the device.
- Added support for dynamic length certificate serial numbers in atcacert.
- Added atcab_write() for lower level write commands.
- Fixed atcah_write_auth_mac(), which had wrong OpCode.
- Added atcab_verify() command for lower level verify commands.
- Added atcab_verify_stored() for verifying data with a stored public key.
- Removed atcab_write_bytes_slot(). Use atcab_write_bytes_zone() instead.
- Modified atcab_write_bytes_zone() and atcab_read_bytes_zone() to specify a

slot

- Added atcab_verify_validate() and atcab_verify_invalidate()
- Improvements to host functions to handle more cases.
- Added atcab_updateextra(), atcab_derive_key()
- Added support for more certificate formats.
- Added general purpose hardware SHA256 functions. See atcab_hw_sha2_256().
- Removed device specific config read/write. Generic now handles both.
- Removed unnecessary response parameter from lock commands.
- Enhanced and added unit tests.
- Encrypted read and write functions now handle keys with SlotConfig.NoMac set
- atcab_cmp_config_zone() handles all devices now.
- Fixed some edge cases in atcab_read_bytes_zone().
- Updated atSHA() to work with all devices.
- Fixed atcacert_get_device_locs() when using stored sn.

01/08/2016

- New HAL implementations for
 - Single Wire interface for SAMD21 / SAMR21
 - SAMV71 I2C HAL implementation
 - XMega A3Bu HAL implementation
- Added atcab_version() method to return current version string of libary to application
- New Bus and Discovery API
 - returns a list of ATCA device configurations for each CryptoAuth device found
 - currently implemented on SAMD21/R21 I2C, SAMV71
 - additional discovery implementations to come
- TLS APIs solidified and documented
- Added missing doxygen documentation for some CryptoAuthLib methods
- Stubs for HAL SPI removed as they are unused for SHA204A and ECC508A support
- bug fixes
- updated atcab_sha() to accept a variable length message that is > 64 bytes and not a multiple of 64 bytes (the SHA block size).
- refactored Cert I/O and Cert Data tests to be smaller
- 'uncrustify' source formatting
- published on GitHub

9/19/2015

- Kit protocol over HID on Windows
- Kit protocol over CDC on Linux
- TLS integration with ATECC508A
- Certificate I/O and reconstruction
- New SHA2 implementation
- Major update to API docs, Doxygen files found in cryptoauthlib/docs
- load cryptoauthlib/docs/index.html with your browser

Host Device Support

CryptoAuthLib will run on a variety of platforms from small micro-controllers to desktop host systems. The current list of hardware abstraction layer support includes:

Rich OS Hosts:

- Linux Kit Protocol over CDC USB
- Linux Kit Protocol over HID USB
- Linux I2C protocol.
- Windows Kit Protocol over CDC USB
- Windows Kit Protocol over HID USB

Microcontrollers:

- SAMD21 (I2C, SWI, and Bit Banging)
- SAMR21 (I2C and SWI)
- SAM4S (I2C)
- SAMV71 (I2C)
- SAMB11 (I2C)
- SAMG55 (I2C)
- AVR XMEGA A3BU (I2C and SWI)
- AVR AT90USB1287 (I2C and SWI)
- PIC32MX695F512H (I2C)

If you have specific microcontrollers or Rich OS platforms you need support for, please contact us through the Microchip portal with your request.

CryptoAuthLib Architecture

See the 'docs' directory of CryptoAuthLib for supporting documentation including architecture diagrams and more detailed usage docs.

The library is structured to support portability to:

- multiple hardware/microcontroller platforms
- multiple environments including bare-metal, Windows, and Linux OS
- multiple chip communication protocols (I2C, SPI, UART, and SWI)

All platform dependencies are contained within the HAL (hardware abstraction layer).

There are three primary object types in CryptoAuthLib:

- Device (ATCADevice)
- Command (ATCACmd)
- Interface (ATCAIface)

ATCADevice is a composite object made up of ATCACmd ATCAIface.

Currently, the vast majority of testing has been performed on:

- ATSAMR21 Xplained Pro
- ATSAMD21 Xplained Pro
- ATSAMV71 Xplained Pro
- Windows (kit protocol HID)

These host containers implement a host test environment and test console to exercise tests. They presume that a CryptoAuth Xplained Pro or other I2C socket for an ATECC608A/ATECC508A/ATECC108A/ATSHA204A are

connected to the I2C pins of the host Xplained Pro development board or in the case of windows is using a HID connection to an ATCK101 or ATCK590.

The unit tests and basic tests exercise the core datasheet commands of the device as well as the more convenient, basic API methods.

If you need an example of how to use a command, these hosts and tests are a good place to reference.

Object Architecture

Even though this is a C library, it follows object-oriented design patterns.

An object is minimally defined to be data and the actions which operate on that data.

Each CryptoAuth device is a composite object, a structure which includes the command table (list of commands) which are valid for the device, and the data used to hold the state of that device.

ATCADevice is the object which represents the Microchip CryptAuth device

ATCACmd is the object which represents the valid methods of the Device.

ATCAInterface is the physical interface object (I2C or SWI instance).

Currently, each Device may have a single OATCAInterface.

ATCADevice represents an ATSHA or ATECC family device.

In order to add new protocol support for a platform, you provide a HAL (hardware abstraction layer) C file for the protocol and target. In your project's IDE or Makefile, you select which HAL support you need for the hardware configuration. Generally, there are separate files for each protocol and platform combination - (ie: samd21_i2c_asf.c would target SAMD21 MCUs with I2C using the ASF low-level driver support.)

Directory Structure

```
docs - AppNotes and Doxygen HTML documentation for the library API. Load "docs/html/index.html" in your browser
lib - primary library source code
lib/atcacert - certificate data and i/o methods
lib/basic - the Basic API way to access the core classes
lib/crypto - Software crypto implementations (primarily SHA1 and SHA256)
lib/hal - hardware abstraction layer code for supporting specific platforms
lib/host - support functions for common host-side calculations
lib/jwt - json web token functions
test - unit tests. See test/cmd-processor.c for main() implementation.
```

For production code, `test` directories should be excluded by not compiling it into a project, so it is up to the developer to include or not as needed. Test code adds significant bulk to an application - it's not intended to be included [in](#) production code.

Tests

There is a set of unit tests found in the test directory which will at least partially demonstrate the use of the objects. Some tests may depend upon a certain device being configured in a certain way and may not work for all devices or specific configurations of the device.

The test/cmd-processor.c file contains a main() function for running the tests. It implements a command-line interface. Typing help will bring up the list of commands available.

One first selects a device type, with one of the following commands:

- 204 (ATSHA204A)
- 108 (ATECC108A)
- 508 (ATECC508A)
- 608 (ATECC608A)

From there the following unit test sweets are available:

- unit (test command builder functions)
- basic (test basic API functions)
- cio (test certification i/o functions)
- cd (test certificate data functions)
- util (test utility functions)
- crypto (test software crypto functions)

Unit tests available depend on the lock level of the device. The unit tests won't lock the config or data zones automatically to allow retesting at desired lock levels. Therefore, some commands will need to be repeated after locking to exercise all available tests.

Starting from a blank device, the sequence of commands to exercise all unit tests is:

```
unit
basic
lockcfg
unit
basic
lockdata
unit
basic
cio
cd
util
crypto
```

Using CryptoAuthLib (Microchip CryptoAuth Library)

Using a new library is often easier when you can load an example and see how it works. We've provided examples in the form of "host containers" which are host projects that incorporate CryptoAuthLib and target various processors or communication APIs.

We maintain host test containers for each of the HAL layers we support. We've published the host container for SAMD21 which demonstrates a simple console interface to invoke test runners.

Look for SAMD21 Unit Tests CryptoAuthLib at
<http://www.microchip.com/SWLibraryWeb/product.aspx?product=CryptoAuthLib>

The best way to learn how to use CryptoAuthLib is to study the host test projects that exercise the library and ATECC and ATSHA devices.

New examples will be forthcoming as the software matures. Continue checking

the [CryptoAuthentication](#)

web page for new updates.

Using Git to Incorporate CryptoAuthLib as a Submodule

You can include this project in your own project under git.

Using CryptoAuthLib as a git submodule, you can maintain your application separately from CryptoAuthLib.

If your project is already in git but you haven't yet intergrated

CryptoAuthLib, change to the directory where you want to put CryptoAuthLib

```
git submodule add -b master <giturl to CryptoAuthLib>
```

This adds CryptoAuthLib as a subdirectory and separate git repo within your own project. Changes and commits to your project vs CryptoAuthLib will remain separated into each respective repository.

If there is a project you want to checkout that already incorporates CryptoAuthLib as a submodule if you clone the repo that incorporates CryptoAuthLib, after cloning, you'll still need to fill out the CryptoAuthLib submodule after cloning:

```
git submodule init  
git submodule update --remote  
cd cryptoauthlib  
git checkout master
```

Now that CryptoAuthLib is a full-fledged submodule in your git project, in order to easily add it to your project within Atmel Studio, please see this [tip](#)

Incorporating CryptoAuthLib in a project

1. In your Makefile or IDE, choose the HAL support you need from the HAL directory and exclude other HAL files from your project.
2. For I2C interfaces, define the symbol ATCA_HAL_I2C in your compiler's symbol definitions. This will hook up the CryptoAuthLib interface class with your HAL implementation of I2C methods.
3. HAL implementations for CDC and HID interfaces to the ATCK101 are also included for use with Windows or Linux versions of the test host.

Incorporating CryptoAuthLib in a Linux project using USB HID devices

The Linux HID HAL files use the Linux udev development software package.

To install the udev development package under Ubuntu Linux, please type the following command at the terminal window:

```
sudo apt-get install libudev-dev
```

This adds the udev development development software package to the Ubuntu Linux installation.

The Linux HID HAL files also require a udev rule to be added to change the

permissions of the USB HID Devices. Please add a new udev rule for the Microchip CryptoAuth USB devices.

```
cd /etc/udev/rules.d  
sudo touch mchp-cryptoauth.rules
```

Edit the mchp-cryptoauth.rules file and add the following line to the file:

```
SUBSYSTEM=="hidraw", ATTRS{idVendor}=="03eb", ATTRS{idProduct}=="2312", MODE=="0666"  
`# ATCA support for Mongoose OS
```

This is a HAL implementation for the Atmel/Microchip CryptoAuthLib. It translates ATCA interface calls into MGOS API calls. Currently only I2C is implemented.

```
# Embedded JavaScript engine  
<p class="mume-header" id="embedded-javascript-engine"></p>
```

This library brings an [mJS: restricted JavaScript-like Engine](<https://github.com/cesanta/mjs>).

Apart from adding the mJS itself, the library creates a global instance of it (available via `mgos_mjs_get_global()`), and also brings a number of mgos-specific API files, see `fs` directory.

```
# How to contribute
```

```
<p class="mume-header" id="how-to-contribute-2"></p>
```

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

```
## Contributor License Agreement
```

```
<p class="mume-header" id="contributor-license-agreement-2"></p>
```

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](<https://cla.developers.google.com/about/google-individual>).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](<https://cla.developers.google.com/about/google-corporate>).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

```
## Submitting a patch
```

```
<p class="mume-header" id="submitting-a-patch-2"></p>
```

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
1. Follow the normal process of [forking](<https://help.github.com/articles/fork-a-repo>) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
1. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing

tests if you're unsure how to go about it.

1. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](<https://github.com/mbrukman/autogen>) tool.
1. Do your best to have [well-formed commit messages](<http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html>) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.

1. Finally, push the commits to your fork and submit a [pull request](<https://help.github.com/articles/creating-a-pull-request>).

VEML6075 I2C Driver

```
<p class="mume-header" id="veml6075-i2c-driver"></p>
```

A Mongoose library for Vishay Technologies UVA/UVB intensity sensor.

Implementation details

```
<p class="mume-header" id="implementation-details-1"></p>
```

The VEML6075 sensor can be configured in continuous or one-shot measurement modes, and upon each measurement returns the 16-bit raw counters for UVA, UVB, visual and IR bands. Using conversion coefficients, the UVIndex can be returned using these data points.

An accurate VEML6075 UVI sensing system requires visible and infrared noise compensation and a teflon diffusor for cosine angular response correction. The UVI formulas and related UVI formula coefficients are discussed here in detail.

This library implements the application [notes](<https://www.vishay.com/docs/84339/designingveml6075.pdf>)

Limitations

```
<p class="mume-header" id="limitations"></p>
```

The sensor measures for a configurable interval and integrates the total light received for each band. Although its sensitivity can easily be adjusted with selecting the proper integration times (from 50ms through to 800ms), the algorithm provided in the Vishay paper is only applicable to 100ms integration times. Therefore, the library currently does not support other integration times.

Example application

```
<p class="mume-header" id="example-application-2"></p>
```

An example program using a timer to read data from the sensor every 5 seconds:

```
#include "mgos.h"
#include "mgos_veml6075.h"

static struct mgos_veml6075 *s_veml6075;

static void timer_cb(void *user_data) {
    float uva, uvb, uvindex;

    uva=mgos_veml6075_getUVA(s_veml6075);
    uvb=mgos_veml6075_getUVB(s_veml6075);
    uvindex=mgos_veml6075_getUVIndex(s_veml6075);

    LOG(LL_INFO, ("VEML6075 uva=%.1f uvb=%.1f uvindex=%.2f", uva, uvb, uvindex));

    (void) user_data;
}
```

```
enum mgos_app_init_result mgos_app_init(void) {
s_veml6075=mgos_veml6075_create(0x10); // Default I2C address
if (s_veml6075) {
mgos_set_timer(5000, true, timer_cb, NULL);
} else {
LOG(LL_ERROR, ("Could not initialize sensor"));
}
return MGOS_APP_INIT_SUCCESS;
}
```

```
# Disclaimer
<p class="mume-header" id="disclaimer-2"></p>
```

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

```
# Arduino Adafruit HTU21D(D) library for Mongoose OS
<p class="mume-header" id="arduino-adafruit-htu21dd-library-for-mongoose-os"></p>
```

This is a port of the [Adafruit Arduino driver for the HTU21D(F) sensor](https://github.com/adafruit/Adafruit-HTU21DF_Library) ported to run on the [Mongoose OS](<https://github.com/mongoose-os-libs/arduino-adafruit-htu21df>).

Usage is extremely simple....

in `mos.yml`, add to `libs:` section,

```
` - origin: https://github.com/mongoose-os-libs/arduino-adafruit-htu21df
```

in your `init.js`, add something like the following,

```
```javascript
load('api_arduino_htu21df.js');
```

and

```
// Initialize Adafruit-HTU21DF library
let htu = Adafruit-HTU21DF.create();
htu.begin();
let htuGetData = function () {
 print('HTU21D: T: ', htu.readTemperature() , 'C RH: ', htu.readHumidity(), '%');
};
let htuTimer = Timer.set(10000 /* milliseconds */, true /* repeat */, htuGetData, null);
```

to use the library.

Enjoy!

## How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

## Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the

project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

## Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

## MAX7219/MAX7221 SPI Driver

A Mongoose library for MAX7219, an 8-digit LED Display Driver.

## Implementation details

The chip is straight forwardly implemented, and allows for daisy chaining multiple devices (by tying `DOUT` of the first chip to `DIN` of the second chip, while connecting `CS`, `CLK`, `VCC` and `GND` to all chips).

After initialization, bytes can be written to so called `digits`, which are the individual lines of an 8x8 LED display, or the actual digits of a 7-segment display.

**NOTE:** The MAX7219 is a 5V device, and as such level shifters are necessary between the `MOSI`, `SCLK` and `CS` pins of the MCU and the pins on the chip(s). It may work without, but it's not recommended!

# API Description

Create the object with `mgos_max7219_create()`, and if there are multiple devices connected, set the correct number with `mgos_max7219_set_num_devices()`. When you're done using the display(s), call `mgos_max7219_destroy()` to turn them off and clean them up.

**`mgos_max7219_set_intensity()`** Intensity of all displays will be set, where the argument is between 0 (very dim) to 15 (very bright).

**`mgos_max7219_set_mode()`** Setting the `mode` is done with a boolean mode argument. If `true` is specified, all connected devices are set into what is called `Code B decoding`, in which the values map internally to segments of a 7-segment display, for example writing 0 will light up the segments needed to draw a 0 on a 7-segment display. There are only 15 values in this system, see the header file for details. The mode is initialized as `false`, in other words do not use `Code B`, and now each bit on the 8-bit values correspond to one LED, typically on an 8x8 LED display.

Writes can be performed in three ways:

**`mgos_max7219_write_raw()`** writes an 8-bit value to a certain line (`digit`) of a certain device (`deviceno`). This will only work if the `mode` was set to `false`.

**`mgos_max7219_write_digit()`** writes a number to a certain position (`digit`) of a certain device (`deviceno`). This will only work if the `mode` was set to `true`.

**`mgos_max7219_write_line()`** writes an array of bytes, which must be of exactly `num_devices` length, to the given line (`digit`) of all connected displays.

## Example application

```

#include "mgos.h"
#include "mgos_config.h"
#include "mgos_max7219.h"

const uint8_t program[] = {
 0, 153, 90, 60, 24, 24, 36, 66, 5,
 0, 24, 24, 255, 24, 24, 36, 66, 5,
 0, 24, 24, 60, 90, 153, 36, 66, 5,
 0, 24, 24, 255, 24, 24, 36, 66, 5,
};

void timer_cb(void *data) {
 struct mgos_max7219 *d = (struct mgos_max7219 *)data;
 static uint8_t waiting = 0;
 static uint16_t position = 0;

 if (!d) return;

 if (waiting > 0) {
 waiting--;
 return;
 }

 waiting = program[position * 9 + 8];
 LOG(LL_INFO, ("Writing program position %u, then waiting %.1f seconds", position, 0.1 * waiting));

 for (int i = 0; i < 8; i++) {
 mgos_max7219_write_raw(d, 0, i, program[position * 9 + i]);
 mgos_max7219_write_raw(d, 1, i, ~(program[position * 9 + i]));
 }

 position++;
 if (position == sizeof(program) / 9) position = 0;
}

enum mgos_app_init_result mgos_app_init(void) {
 struct mgos_max7219 *d = NULL;

 if (!(d = mgos_max7219_create(mgos_spi_get_global(), mgos_sys_config_get_max7219_cs_index()))) {
 LOG(LL_ERROR, ("Could not create MAX7219 display"));
 return MGOS_APP_INIT_ERROR;
 }
 mgos_max7219_set_num_devices(d, 2);
 mgos_set_timer(100, true, timer_cb, d);

 return MGOS_APP_INIT_SUCCESS;
}

```

## Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

## A Texas Instruments TMP006 temperature sensor

### Overview

This is a library for the Texas Instruments TMP006 temperature sensor.

See <http://www.ti.com/product/TMP006> for more information about the hardware.

# Support library for the device management dashboard

# Overview

This library provides support for device management dashboard.

# Implementation of Mongoose OS RPC over UDP

## Overview

Frames are sent as datagrams. Fragmentation is not supported, so 1 frame = 1 datagram.

Responses are supported, but because UDP is not reliable, it is best suited for notifications.

## Listener configuration

Set `rpc.udp.listen_addr` to `udp://0.0.0.0:1234` to listen on port 1234.

## Outbound channel configuration

To use UDP for outbound RPC, set `rpc.udp.dst_addr` to the address of the server / peer, e.g. `udp://192.168.1.23:1234`.

## mos tool support

`mos` supports UDP transport via the `udp://` port scheme, e.g.:

```
$ mos call --port udp://192.168.11.5:1234/ Sys.GetInfo
```

## How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

## Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

# Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

# Acknowledgements

This code was inspired by existing work by the following entities:

## Adafruit Industries

Portions of this code are taken from Adafruit's STMPE610 library:

- The initialization sequence and defines in `include/mgos_stmpe610.h`

Adafruit licensed their code under the [MIT License](#).

# STMP610E

Mongoose native SPI driver for STMP610E

## Introduction

STMPE610 is a chip that drives a resistive touchscreen. It measures pressure on a glass plate which has an X and Y resistor samples the pressure put on the screen (typically by pressing it).

## API

This driver initializes the chip and allows the user to register a callback, which will be called each time the driver senses a touch (`TOUCH_DOWN`) or a release (`TOUCH_UP`). The callback function will be given the X and Y coordinates, together with an average pressure (Z). For `TOUCH_DOWN` events, the `length` field is set to 1. For `TOUCH_UP` events, the `length`

field is set to the amount of samples read, between 1 and 128.

The X and Y coordinates as read by the screen are between [0..4095), so a mapping function is applied by `mgos_stmpe610_set_dimensions()`. For screens that are rotated, `mgos_stmpe610_set_rotation()` sets the mapping function to return them in the correct orientation.

## Details

The chip uses an interrupt pin to signal the micro controller that it has something to say. The driver sets up a GPIO pin with a pullup, which the chip pulls to ground to initiate the interrupt. Mongoose OS will receive this and issue a software callback (so no hardware interrupts are used), and the driver will clear interrupts on the STMPE610 after the user supplied handler is called.

Users can query the driver to see if the chip is currently registering a touch by calling `mgos_stmpe610_is_touching()` which will return `true` if the user is currently touching the screen.

If the user touches the screen very briefly (only one sample cycle of the driver), the callback will be called with `TOUCH_DOWN`, but no `TOUCH_UP` event will be generated. To work around this, the driver sets a 100ms timer upon `TOUCH_DOWN`, and if the screen is not being touched when the timer expires, a gratuitous `TOUCH_UP` event is sent with `length` set to 1.

## Example Application

### mos.yml

The driver uses the Mongoose native SPI driver. It is configured by setting up the `MOSI`, `MISO`, `SCLK` pins and assinging one of the three available `CS` positions, in this example (which was taken from the Huzzah32 ESP32 microcontroller), we are going to use `CS0`:

```
config_schema:
 - ["spi.enable", true]
 - ["spi.cs0_gpio", 32] # The STMPE610 CS pin
 - ["spi.cs1_gpio", -1]
 - ["spi.cs2_gpio", -1]
 - ["spi.mosi_gpio", 18]
 - ["spi.miso_gpio", 19]
 - ["spi.sclk_gpio", 5]
 - ["stmpe610.irq_pin", 23]
 - ["stmpe610.cs_index", 0] # Use spi.cs0_gpio
```

## Application

```

#include "mgos.h"
#include "mgos_stmpe610.h"

static void touch_handler(struct mgos_stmpe610_event_data *ed) {
 if (!ed) return;

 LOG(LL_INFO, ("Touch %s at (%d,%d) pressure=%d, length=%d",
 ed->direction==TOUCH_UP?"UP":"DOWN", ed->x, ed->y, ed->z, ed->length));
}

enum mgos_app_init_result mgos_app_init(void) {
 mgos_stmpe610_set_handler(touch_handler);
 mgos_stmpe610_set_rotation(STMPE610_LANDSCAPE);
 mgos_stmpe610_set_dimensions(320, 240);

 return MGOS_APP_INIT_SUCCESS;
}

```

# Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

## A short guide to use fontconvert.c to create your own fonts using MinGW.

### STEP 1: INSTALL MinGW

Install MinGW (Minimalist GNU for Windows) from [MinGW.org](http://MinGW.org).

Please read carefully the instructions found on [Getting started page](#).

I suggest installing with the "Graphical User Interface Installer".

To complete your initial installation you should further install some "packages".

For our purpose you should only install the "Basic Setup" packages.

To do that:

1. Open the MinGW Installation Manager
2. From the left panel click "Basic Setup".
3. On the right panel choose "mingw32-base", "mingw-gcc-g++", "mingw-gcc-objc" and "msys-base" and click "Mark for installation"
4. From the Menu click "Installation" and then "Apply changes". In the pop-up window select "Apply".

### STEP 2: INSTALL Freetype Library

To read about the freetype project visit [freetype.org](http://freetype.org).

To Download the latest version of freetype go to [download page](#)

and choose "freetype-2.7.tar.gz" file (or a newer version if available).

To avoid long cd commands later in the command prompt, I suggest you unzip the file in the C:\ directory.

(I also renamed the folder to "ft27")

Before you build the library it's good to read these articles:

- [Using MSYS with MinGW](#)
- [Installation and Use of Supplementary Libraries with MinGW](#)
- [Include Path](#)

Inside the unzipped folder there is another folder named "docs". Open it and read the INSTALL.UNIX (using notepad).

Pay attention to paragraph 3 (Build and Install the Library). So, let's begin the installation.

To give the appropriate commands we will use the MSYS command prompt (not cmd.exe of windows) which is UNIX like.

Follow the path C:\MinGW\msys\1.0 and double click "msys.bat". The command prompt environment appears.

Enter "ft27" directory using the cd commands:

```
cd /c
cd ft27
```

and then type one by one the commands:

```
./configure --prefix=/mingw
make
make install
```

Once you're finished, go inside "C:\MinGW\include" and there should be a new folder named "freetype2".

That, hopefully, means that you have installed the library correctly !!

### STEP 3: Build fontconvert.c

Before proceeding I suggest you make a copy of Adafruit\_GFX\_library folder in C:\ directory.

Then, inside "fontconvert" folder open the "makefile" with an editor ( I used notepad++).

Change the commands so in the end the program looks like :

```
all: fontconvert

CC = gcc
CFLAGS = -Wall -I c:/mingw/include/freetype2
LIBS = -lfreetype

fontconvert: fontconvert.c
 $(CC) $(CFLAGS) $< $(LIBS) -o $@

clean:
 rm -f fontconvert
```

Go back in the command prompt and with a cd command enter the fontconvert directory.

```
cd /c/adafruit_gfx_library/fontconvert
```

Give the command:

```
make
```

This command will, eventually, create a "fontconvert.exe" file inside fontconvert directory.

### STEP 4: Create your own font header files

Now that you have an executable file, you can use it to create your own fonts to work with Adafruit GFX lib.

So, if we suppose that you already have a .ttf file with your favorite fonts, jump to the command prompt and type:

```
./fontconvert yourfonts.ttf 9 > yourfonts9pt7b.h
```

You can read more details at: [learn.adafruit](#).

Taraaaaaammm !! you've just created your new font header file. Put it inside the "Fonts" folder, grab a cup of coffee and start playing with your Arduino (or whatever else ....)+ display module project.

## Cesanta note

# Adafruit GFX Library

This is the core graphics library for all our displays, providing a common set of graphics primitives (points, lines, circles, etc.). It needs to be paired with a hardware-specific library for each display device we carry (to handle the lower-level functions).

Adafruit invests time and resources providing this open source code, please support Adafruit and open-source hardware by purchasing products from Adafruit!

Written by Limor Fried/Ladyada for Adafruit Industries.

BSD license, check license.txt for more information.

All text above must be included in any redistribution.

Recent Arduino IDE releases include the Library Manager for easy installation. Otherwise, to download, click the DOWNLOAD ZIP button, uncompress and rename the uncompressed folder Adafruit\_GFX. Confirm that the Adafruit\_GFX folder contains Adafruit\_GFX.cpp and Adafruit\_GFX.h. Place the Adafruit\_GFX library folder your /Libraries/ folder. You may need to create the Libraries subfolder if it's your first library. Restart the IDE.

## Useful Resources

- **Image2Code:** This is a handy Java GUI utility to convert a BMP file into the array code necessary to display the image with the drawBitmap function. Check out the code at ehubin's GitHub repository: <https://github.com/ehubin/Adafruit-GFX-Library/tree/master/Img2Code>
- **drawXBitmap function:** You can use the GIMP photo editor to save a .xbm file and use the array saved in the file to draw a bitmap with the drawXBitmap function. See the pull request here for more details: <https://github.com/adafruit/Adafruit-GFX-Library/pull/31>
- 'Fonts' folder contains bitmap fonts for use with recent (1.1 and later) Adafruit\_GFX. To use a font in your Arduino sketch, #include the corresponding .h file and pass address of GFXfont struct to setFont(). Pass NULL to revert to 'classic' fixed-space bitmap font.
- 'fontconvert' folder contains a command-line tool for converting TTF fonts to Adafruit\_GFX.h format.

## Arduino Adafruit GFX library for Mongoose OS

## RPC support for Google Cloud Platform

### Device support

This library provides RPC support over the mechanisms provided by the Google IoT Core platform

- Requests should be sent to the device as **commands**.  
This library listens to commands on a particular subfolder (default: `rpc`) and parses the payload as RPC frames.
- Responses are published as telemetry events, to the subfolder that is equal to the `dst` of the response (and thus `src` in the original request).

So, if the caller wishes to receive response on subfolder `foo`, they should specify `src: "foo\\"` in their request.

It should be noted that responses from all the devices that use the same response subfolder end up in the same topic and subscription.

All the subscribers will receive messages posted here by all the devices. Subscribers need to take care to not ack messages that are not intended for them (match request ID).

While acceptable for light use, this will cause unnecessary churn if multiple devices are communicating at the same time.

In this case users should set up use multiple different response subfolders.

To receive responses, caller must set up the necessary PubSub topics and/ subscription and ensure that the desired subfolder is plumbed to the correct topic.

See [here](#) for detailed description.

## Mos tool support

mos supports GCP RPC mechanism through the gcp "port" type:

```
$ mos --port gcp://project/region/registry/device call Sys.GetInfo
{
 "app": "demo-c",
 "fw_version": "1.0",
 "fw_id": "20190123-121047/2.10.0-244-g81233d5ed-dirty-mos8",
 "mac": "1AFE34A5930F",
 "arch": "esp8266",
 "uptime": 5248,
 "ram_size": 51864,
 "ram_free": 37072,
 "ram_min_free": 21928,
 "fs_size": 233681,
 "fs_free": 153612,
 "wifi": {
 "sta_ip": "192.168.11.25",
 "ap_ip": "",
 "status": "got ip",
 "ssid": "Over_9000_Internets"
 }
}
```

Use your own project, region, registry, and device.

By default, mos uses rpc subfolder for telemetry events and subscribes to the rpc topic.

You will need to create the topic and forward the rpc subfolder to it.

Or allow mos to set up all the necessary plumbing by adding --gcp-create-topic (Note: config changes to GCP take some time to propagate, so the first request may fail).

You can customize subfolder names via query parameters (e.g. mos --port=gcp://project/region/registry/device?sub=rpc1&respsf=rpc1):

- sub - subscription name. Default is rpc.
- topic - topic name. Default is rpc.
- reqsf - request subfolder. This needs to match rpc.gcp.subfolder on the device. Default is rpc.
- respsf - response subfolder. Default is rpc.
  - As described above, there is inherent contention between subscribers when the same response subfolder is used for responses to requests by multiple callers.  
If the volume of churn (Nacks) becomes a problem, and until Google implements [this](#) or [this](#), the workaround is to set up several response subfolders and shard responses between them.

## Library for modular scrolling LED matrix text displays

[Version 1.0 Video](#)

[Version 2.0 Video](#)

[Library Documentation](#)

Parola is a modular scrolling text display using MAX7219 or MAX7221 LED matrix display controllers using Arduino. The display is made up of any number of identical modules that are plugged together to create a wider/longer display.

- Text left, right or center justification in the display
- Text scrolling, entry and exit effects
- Control display parameters and animation speed
- Support for hardware SPI interface

- Multiple virtual displays (zones) in each string of LED modules
- User defined fonts and/or individual characters substitutions
- Support for double height displays

The aim was to create a 'lego-like' LED matrix display, using standard 8x8 LED matrices. The software supports this flexibility through a scalable approach that only requires the definition of the number of modules to adapt existing software to a new configuration.

The Parola software has a dependency on the [MD\\_MAX72xx Arduino library](#) which implements hardware functions of the LED matrix. The library needs to be configured for the type of matrices being used - please refer to the hardware section of documentation for the [MD\\_MAX72xx library](#).

Parola discussion on the [Arduino forum](#) and kits available from [ElectroDragon](#).

Additional information also at [my blog](#).

## How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

## Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

## Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the

[autogen](#) tool.

5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

# Mongoose OS IMU Library

This library provides a simple API that describes inertial measurement units. It implements various popular I2C and SPI IMUs behind that API. Callers of the library can rely on the API returning consistently typed data regardless of the choice of sensor.

## Anatomy

This library provides an API to expose various implementations of gyroscopes, accelerometers, and magnetometers. It offers a suitable abstraction that presents the sensor data in a consistent way, so application authors need not worry about the implementation details of the sensors.

Users create a `struct mgos_imu` object, to which they add sensors (gyroscope, accelerometer and magnetometer) either via I2C or SPI. Some chips have all three sensor types, some have only two (typically accelerometer and gyroscope) and some have only one type. After successfully adding the sensors to the `mgos_imu` object, reads can be performed, mostly by performing `mgos_imu_*_get()` calls.

All implementations offer the calls described below in the [IMU API](#). Some implementations offer chip-specific addendums, mostly setting sensitivity and range options.

## IMU API

### IMU primitives

`struct mgos_imu *mgos_imu_create()` -- This call creates a new (opaque) object which represents the IMU device. Upon success, a pointer to the object will be returned. If the creation fails, `NULL` is returned.

`void mgos_imu_destroy()` -- This cleans up all resources associated with the IMU device. The caller passes a pointer to the object pointer. If any sensors are attached (gyroscope, accelerometer, magnetometer), they will be destroyed in turn using `mgos_imu_*_destroy()` calls.

`bool mgos_imu_read()` -- This call schedules all sensors attached to the IMU to be read from. It is not generally necessary to call this method directly, as the `mgos_imu_*_get()` calls internally schedule reads from the sensors as well.

`bool mgos_imu_accelerometer_present()` -- This returns `true` if the IMU has an attached accelerometer sensor, or `false` otherwise.

`bool mgos_imu_gyroscope_present()` -- This returns `true` if the IMU has an

attached gyroscope sensor, or `false` otherwise.

`bool mgos_imu_magnetometer_present()` -- This returns `true` if the IMU has an attached magnetometer, or `false` otherwise.

## IMU Sensor primitives

For each of **accelerometer**, **gyroscope** and **magnetometer**, the following primitives exist:

`bool mgos_imu_*_create_i2c()` -- This attaches a sensor to the IMU based on the `type` enum given, using the `i2c` bus and specified `i2caddr` address. The function will return `true` upon success and `false` in case either detection of the sensor, or creation of it failed.

`bool mgos_imu_*_create_spi()` -- This attaches a sensor to the IMU based on the `type` enum given, using the `spi` bus and a specified cable select in `cs_gpio`. The function will return `true` upon success and `false` in case either detection of the sensor or creation of it failed.

`bool mgos_imu_*_destroy()` -- This detaches a sensor from the IMU if it exists. It takes care of cleaning up all resources associated with the sensor, and detaches it from the `i2c` or `spi` bus. The higher level `mgos_imu_destroy()` call uses these lower level calls to clean up sensors as well.

`bool mgos_imu_*_get()` -- This call returns sensor data after polling the sensor for the data. It returns `true` if the read succeeded, in which case the floats pointed to by `x`, `y` and `z` are filled in. If an error occurred, `false` is returned and the contents of `x`, `y` and `z` are unmodified. Note the units of the return values:

- **magnetometer** returns units of Gauss.
- **accelerometer** returns units of G.
- **gyroscope** returns units of degrees per second.

`const char *mgos_imu_*_get_name()` -- This returns a symbolic name of the attached sensor, which is guaranteed to be less than or equal to 10 characters and always exist. If there is no sensor of this type attached, `VOID` will be returned. If the sensor is not known, `UNKNOWN` will be returned. Otherwise, the chip manufacturer / type will be returned, for example `MPU9250` or `ADXL345` or `MAG3110`.

`bool mgos_imu_*_set_orientation()` and `bool mgos_imu_*_get_orientation()` -- Often times a 9DOF sensor will have multiple chips, whose axes do not line up correctly. Even within a single chip the accelerometer, gyroscope and magnetometer axes might not line up (for an example of this, see [MPU9250 chapter 9.1](#)).

To ensure that the `x`, `y`, and `z` axes on all sensors are pointed in the same direction, we can set the orientation on the gyroscope and magnetometer. See `mgos_imu.h` for more details and an example of how to do this.

## Supported devices

### Accelerometer

- MPU9250 and MPU9255
- ADXL345

- LSM303D and LSM303DLM
- MMA8451
- LSM9DS1
- LSM6DSL
- MPU6000 and MPU6050
- ICM20948

## Gyroscope

- MPU9250 and MPU9255
- L3GD20 and L3GD20H
- ITG3205
- LSM9DS1
- LSM6DSL
- MPU6000 and MPU6050
- ICM20948

## Magnetometer

- MAG3110
- AK8963 (as found in MPU9250/MPU9255)
- AK8975
- LSM303D and LSM303DLM
- HMC5883L
- LSM9DS1
- ICM20948

## Adding devices

This is a fairly straight forward process, taking the `ADXL345` accelerometer as example:

1. Add a new driver header and C file, say `src/mgos_imu_adxl345.[ch]`. The header file exposes all of the `#define`'s for registers and constants.
2. In the header file, declare a `detect`, `create`, `destroy` and `read` function.
  - `bool mgos_imu_adxl345_detect()` -- this function optionally attempts to detect the chip, often times by reading a register or set of registers which uniquely identifies it. Not all chips can actually be detected, so it's OK to not define this function at all.
  - `bool mgos_imu_adxl345_create()` -- this function has to perform the initialization of the chip, typically by setting the right registers and possibly creating a driver-specific memory structure (for, say, coefficients or some such). If used, that memory structure is attached to the `user_data` pointer, and if so, the implementation of the `_destroy()` function must clean up and free this memory again.
  - `bool mgos_imu_adxl345_read()` -- this function performs the chip specific read functionality. This will be called whenever the user asks for data, either by calling `mgos_imu_read()` or by calling `mgos_imu_*_get()`.
  - `bool mgos_imu_adxl345_destroy()` -- this function deinitializes the chip, and optionally clears and frees the driver-specific memory structure in `user_data`. Not all chips need additional memory structures or deinitialization code, in which case it's OK to not define this function at all.

3. Implement the `detect`, `create`, `destroy` and `read` functions in the source code file `src/mgos_imu_adxl345.c`.
4. Add the device to one of the `enum mgos_imu_*_type` in `include/mgos_imu.h`. In the example case of adxl345 (which is an accelerometer), add it to `enum mgos_imu_acc_type`.
5. Add a string version of this to function `mgos_imu_*_get_name()` so that callers can resolve the sensor to a human readable format. Make sure that the string name is not greater than 10 characters.
6. Add the type to the `switch()` in `mgos_imu_*_create_i2c()` (or `_spi()`) functions.
7. Update this document to add the driver to the list of supported drivers.
8. Test code on a working sample, and send a PR using the guidelines laid out in [contributing](#).

It is important to note a few implementation rules when adding drivers:

- New drivers MUST NOT change any semantics of the abstraction layer (`mgos_imu` and `mgos_imu_*` members) nor the `mgos_imu_*_get()` functions.
- Implementations of drivers MUST provide bias, scaling and other normalization in the driver itself. What this means, in practice, is that the correct units must be produced (`m/s/s`, `deg/s` and `uT`).
- Pull Requests MUST NOT mix driver and abstraction changes. Separate them.
- Changes to the abstraction layer MUST be proven to work on all existing drivers, and will generally be scrutinized.

## Example driver (AK8975)

Here's an example, for the magnetometer chip AK8975, showing a set of commits for each of the steps above (and honoring the driver implementation rules).

1. Add `src/mgos_imu_ak8975.[ch]` [commit](#)
2. Extend `enum mgos_imu_mag_type` in `include/mgos_imu.h` [commit](#)
3. Add to `mgos_imu_magnetometer_get_name()` in `src/mgos_imu_magnetometer.c` [commit](#)
4. Add to `mgos_imu_magnetometer_create_i2c()` in `src/mgos_imu_magnetometer.c` [commit](#)

## Example Code

An example program that creates an IMU of type `MPU9250` (which has an accelerometer, a gyroscope and a magnetometer all in one tiny package):

```

#include "mgos.h"
#include "mgos_i2c.h"
#include "mgos_imu.h"

static void imu_cb(void *user_data) {
 struct mgos_imu *imu = (struct mgos_imu *)user_data;
 float ax, ay, az;
 float gx, gy, gz;
 float mx, my, mz;

 if (!imu) return;

 if (mgos_imu_accelerometer_get(imu, &ax, &ay, &az))
 LOG(LL_INFO, ("type=%-10s Accel X=%.2f Y=%.2f Z=%.2f",
 mgos_imu_accelerometer_get_name(imu), ax, ay, az));
 if (mgos_imu_gyroscope_get(imu, &gx, &gy, &gz))
 LOG(LL_INFO, ("type=%-10s Gyro X=%.2f Y=%.2f Z=%.2f",
 mgos_imu_gyroscope_get_name(imu), gx, gy, gz));
 if (mgos_imu_magnetometer_get(imu, &mx, &my, &mz))
 LOG(LL_INFO, ("type=%-10s Mag X=%.2f Y=%.2f Z=%.2f",
 mgos_imu_magnetometer_get_name(imu), mx, my, mz));
}

enum mgos_app_init_result mgos_app_init(void) {
 struct mgos_i2c *i2c = mgos_i2c_get_global();
 struct mgos_imu *imu = mgos_imu_create();
 struct mgos_imu_acc_opts acc_opts;
 struct mgos_imu_gyro_opts gyro_opts;
 struct mgos_imu_mag_opts mag_opts;

 if (!i2c) {
 LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
 return false;
 }

 if (!imu) {
 LOG(LL_ERROR, ("Cannot create IMU"));
 return false;
 }

 acc_opts.type = ACC_MPU9250;
 acc_opts.scale = 16.0; // G
 acc_opts.odr = 100; // Hz
 if (!mgos_imu_accelerometer_create_i2c(imu, i2c, 0x68, &acc_opts))
 LOG(LL_ERROR, ("Cannot create accelerometer on IMU"));

 gyro_opts.type = GYRO_MPU9250;
 gyro_opts.scale = 2000; // deg/sec
 gyro_opts.odr = 100; // Hz
 if (!mgos_imu_gyroscope_create_i2c(imu, i2c, 0x68, &gyro_opts))
 LOG(LL_ERROR, ("Cannot create gyroscope on IMU"));

 mag_opts.type = MAG_AK8963;
 mag_opts.scale = 12.0; // Gauss
 mag_opts.odr = 10; // Hz
 if (!mgos_imu_magnetometer_create_i2c(imu, i2c, 0x0C, &mag_opts))
 LOG(LL_ERROR, ("Cannot create magnetometer on IMU"));

 mgos_set_timer(1000, true, imu_cb, imu);
 return true;
}

```

## Demo Code

For a cool demo, take a look at my [Demo Apps](#):

# Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

## SSD1306 OLED Display Driver Support

This library adds support for the Solomon Systech SSD1306 OLED display driver to Mongoose OS.

The default configuration and PIN mapping supports ESP32 boards with the driver hard-wired for I2C communication via GPIO pins 4 and 5. It should work on other platforms, but has not yet been tested.

This driver should support displays of any resolution supported by the SSD1306.

This driver does not yet support SPI operation.

### Note about the [LOLIN / WEMOS OLED shield](#)

It uses a 64x48 panel with column offset of 32, correct configuration for it is as follows:

```
config_schema:
- ["i2c.enable", true]
- ["i2c.sda_gpio", 4]
- ["i2c.scl_gpio", 5]
- ["ssd1306.i2c.enable", false] # Use system bus.
- ["ssd1306.enable", true]
- ["ssd1306.width", 64]
- ["ssd1306.height", 48]
- ["ssd1306.col_offset", 32]
```

## Implementation of Mongoose OS RPC over UART

## Low level Bluetooth support

Currently contains only GATT server implementation for ESP32.

## Configuration section

`bt-common` library adds a `bt` configuration section with the following settings:

```

"bt": {
 "enable": true, // Enabled by default. Disabled on first reboot with WiFi on
 "dev_name": "", // Device name. If empty, value equals to device.id
 "adv_enable": true, // Advertise our Bluetooth services
 "keep_enabled": true, // Keep enabled after successful boot with WiFi on
 "scan_rsp_data_hex": "", // Custom scan response data, as hex string (e.g. `48656c6c6f` for `Hello`)
 "allow_pairing": true, // Allow pairing/bonding with other devices
 "max_paired_devices": 10, // Allow pairing with up to this many devices; -1 - no limit
 "gatts": {
 "min_sec_level": 0, // Minimum security level for all attributes of all services.
 // 0 - no auth required, 1 - encryption reqd, 2 - encryption + MITM reqd
 "require_pairing": false // Require that device is paired before accessing services
 }
}

```

## Security

Default settings allow for unrestricted access: anyone can pair with a device and access the services.

A better idea is to set `bt.gatts.require_pairing` to true, `bt.allow_pairing` to false and only enable it for a limited time via `mgos_bt_gap_set_pairing_enable` when user performs some action, e.g. presses a button.

Raising `bt.gatts.min_sec_level` to at least 1 is also advisable.

*Note:* At present, level 2 (MITM protection) is not usable as it requires device to have at least output capability during pairing, and there's no API for displaying the pairing code yet.

## A DHT Mongoose OS library

### Overview

This is a library for the DHT series of low cost temperature/humidity sensors.  
# Library to read Dallas temperature sensor using ESP32's RMT device

The library uses the `onewire-rmt` library.

A pullup resistor (4.7k) must be connected between the data line of the sensor and Vcc.

### Example code in C++

```

#include <mgos.h>
#include "DallasRmt.h"

static DallasRmt* dallas = NULL;
static OnewireRmt* ow = NULL;
static int numDevices = 0;
static uint32_t readings = 0;

static void loop_onewire(void *arg)
{
 (void) arg;
 if (0 == numDevices) {
 LOG(LL_WARN, ("No device found"));
 return;
 }
 bool wait = dallas->getWaitForConversion();
 double start = mg_time();
 dallas->requestTemperatures();
 double end = mg_time();
 LOG(LL_WARN, ("wait=%d, conversionTime=%.2fms", wait, 1000.0 * (end - start)));

 uint8_t deviceAddress[8];
 for (int i = 0; i < numDevices; ++i) {
 float temp = dallas->getTempCByIndex(i);
 dallas->getAddress(deviceAddress, i);
 char buf[17];
 char*p = buf;
 for (int i = 0; i < 8; ++i) {
 p += snprintf(p, sizeof (buf) - 2 * i, "%02x", deviceAddress[i]);
 }
 LOG(LL_WARN, ("index=%d, address=%s, temp=% .2f, reading=%d", i, buf, temp, readings));
 readings++;
 }
}

extern "C"
enum mgos_app_init_result mgos_app_init(void)
{
 dallas = new DallasRmt();
 ow = new OnewireRmt(13 /*pin*/, 0 /*rmt_rx*/, 1 /*rmt_tx*/);
 dallas->setOneWire(ow);
 dallas->begin();
 numDevices = dallas->getDeviceCount();

 mgos_set_timer(3000, 1, loop_onewire, NULL);

 return MGOS_APP_INIT_SUCCESS;
}

```

## Example code in C

```

#include <mgos.h>
#include "mgos_dallas_rmt.h"

static DallasRmt* dallas = NULL;
static int numDevices = 0;
static uint32_t readings = 0;

static void readData() {
 char deviceAddress[8];
 for (int i = 0; i < numDevices; ++i) {
 float temp = mgos_dallas_rmt_get_tempc_by_index(dallas, i) / 100.0;
 mgos_dallas_rmt_get_address(dallas, deviceAddress, i);
 char buf[17];
 char*p = buf;
 for (int i = 0; i < 8; ++i) {
 p += snprintf(p, sizeof (buf) - 2 * i, "%02x", deviceAddress[i]);
 }
 }
}

```

```

}

LOG(LL_WARN, ("index=%d, address=%s, temp=%.2f, reading=%d", i, buf, temp, readings));
readings++;
}

static void owTimerCB(void * arg) {
 readData();
}

static void loop_onewire(void *arg) {
 (void) arg;
 if (0 == numDevices) {
 LOG(LL_WARN, ("No device found"));
 return;
 }
 bool wait = mgos_dallas_rmt_get_wait_for_conversion(dallas);
 double start = mg_time();
 mgos_dallas_rmt_request_temperatures(dallas);
 double end = mg_time();
 LOG(LL_WARN, ("wait=%d, conversionTime=%.2fms", wait, 1000.0 * (end - start)));
 if (wait) {
 readData();
 } else {
 mgos_set_timer(800, 0, owTimerCB, NULL);
 }
}

enum mgos_app_init_result mgos_app_init(void) {
 OnewireRmt* ow = mgos_onewire_rmt_create(13 /*pin*/, 0 /*rmt_rx*/, 1 /*rmt_tx*/);
 dallas = mgos_dallas_rmt_create(ow);

 mgos_dallas_rmt_begin(dallas);
 numDevices = mgos_dallas_rmt_get_device_count(dallas);

 int resolution = mgos_dallas_rmt_get_global_resolution(dallas);
 int16_t millis = mgos_dallas_rmt_millis_to_wait_for_conversion(dallas, resolution);
 LOG(LL_WARN, ("resolution=%d, millis=%d", resolution, millis));
 //mgos_dallas_rmt_set_wait_for_conversion(dallas, false);

 mgos_set_timer(3000, 1, loop_onewire, NULL);

 return MGOS_APP_INIT_SUCCESS;
}

```

## # WiFi support

This library provides common WiFi API.

### ## Configuration

A library adds a `wifi` configuration entry with two sub-entries: `wifi.ap` which contains configuration settings for the Access Point mode, and `wifi.sta` for the Station mode.

### ### Station configuration

```

```javascript
"wifi": {
    "sta": {
        "enable": true,      // Enable Station mode
        "ssid": "",         // WiFi network name
        "pass": "",         // Password
        "user": "",          // Username for WPA-PEAP mode
        "anon_identity": "", // Anonymous identity for WPA mode
        "cert": "",          // Client certificate for WPA-TTLS mode
        "key": "",           // Client key for WPA-TTLS mode
        "ca_cert": "",       // CA certificate for WPA-enterprise mode
        "ip": "",            // Static IP Address
        "netmask": "",        // Static Netmask
        "gw": ""             // Static Default Gateway
    }
}
```

```

```

"nameserver": "", // DNS Server
"dhcp_hostname": "" // Host name to include in DHCP requests
},
"sta1": {
...
},
"sta2": {
...
},
"sta_cfg_idx": 0, // Station config index to start connecting with, 0, 1 or 2.
"sta_connect_timeout": 30 // Timeout for connection, seconds.
}

```

## Multiple Station Configurations

Station configurations will be tried starting from `sta_cfg_idx` and each one that is enabled will be given `sta_connect_timeout` seconds to connect. Successfully connected station's index will be saved in `sta_cfg_idx` so next boot will start with previously used configuration.

Setting `sta_connect_timeout` to 0 disables this logic.

## Access Point configuration

```

"wifi": {
 "ap": {
 "enable": true, // Enable Access Point mode
 "ssid": "Mongoose_??????", // SSID to use. ?? symbols are substituted by MAC address
 "pass": "Mongoose", // Password
 "hidden": false, // Hide WiFi network
 "channel": 6, // WiFi channel
 "max_connections": 10, // Maximum number of connections
 "ip": "192.168.4.1", // Static IP Address
 "netmask": "255.255.255.0", // Static Netmask
 "gw": "192.168.4.1", // Static Default Gateway
 "dhcp_start": "192.168.4.2", // DHCP Start Address
 "dhcp_end": "192.168.4.100", // DHCP End Address
 "trigger_on_gpio": -1 // Trigger AP on low GPIO
 }
}

```

# IBM Watson IoT Platform support library for Mongoose OS

See [IBM Watson tutorial](#)

## Arduino Wire (I2C) library for Mongoose OS

This library provides an Arduino compatibility layer for the I2C protocol by providing a `Wire.h` public header, so that one could pick an existing Arduino program which uses I2C, throw it into the Mongoose OS application sources, and ideally, it "just works".

This library provides SPI ([Serial Peripheral Interface](#)) support for Mongoose OS.

## Mongoose OS Driver for TI BQ27421 Fuel Gauge

### Overview

## Example usage

Initialize and read some registers:

```
include "mgos.h"
<p class="mume-header" id="include-mgosh-15"></p>

include "mgos_bq27421.h"
<p class="mume-header" id="include-mgos_bq27421h"></p>

struct mgos_bq27421 *g = mgos_bq27421_create(mgos_i2c_get_global());

uint16_t dt = 0, fwv = 0, dcap = 0;

if (!mgos_bq27421_ctl_read(g, MGOS_BQ27421_CTL_DEVICE_TYPE, &dt) ||
 !mgos_bq27421_ctl_read(g, MGOS_BQ27421_CTL_FW_VERSION, &fwv) ||
 !mgos_bq27421_reg_read(g, MGOS_BQ27421_REG_CAP_DESIGN, &dcap)) {
 // Error
}

LOG(LL_INFO, ("Gauge type 0x%u, fw version 0x%u, battery design capacity %u mAh", dt, fwv, dcap));
```

Update battery design capacity (signed 16-bit integer in data class 82 at offset 10):

```
if (!mgos_bq27421_data_write_i2(g, 82, 10, 1234 /* mAh */)) {
 // Error
}
```

The rest of the API is documented in the [include/mgos\_bq27421.h](header file).

## Acknowledgements

This driver is contributed by [Watchdog System](#).

## License

Released under Apache License 2.0, see [LICENSE](#) file.

# Implementation of Mongoose OS RPC over MQTT protocol

## Overview

MQTT RPC channel allows invoking RPC calls via MQTT.

## Topics

When `rpc.mqtt.enable` is true (default), device subscribes to `rpc.mqtt.sub_topic` (default:  `${device.id}/rpc`, e.g. `esp8266_DA7E15/rpc` in the example).

Responses are published to  `${src}/rpc`, where  `${src}` is taken from the request's frame, so requests for which a response is expected must include `src`.

# Examples

## Request

Published to `esp8266_DA7E15/rpc`:

```
{ "id": 123, "src": "foo", "method": "Sys.GetInfo"}
```

Response will be published to `foo/rpc`.

## mos tool support

`mos` tool supports RPC over MQTT via the `mqtt` port schema:

```
$ mos --port mqtt://iot.eclipse.org/esp8266_DA7E15 call Sys.GetInfo
```

For brokers that require TLS, `mqtts` should be used:

```
$ mos --port mqtts://iot.eclipse.org/esp8266_DA7E15 call Sys.GetInfo
```

For brokers that require TLS client authentication (e.g. AWS), cert and file should be supplied:

```
$ mos --port mqtts://XXXXXXXXX.iot.REGION.amazonaws.com/esp8266_DA7E15 --cert-file mycert.pem --key-file mykey.pem call Sys.GetInfo
```

# Bluetooth Current Time Service implementation

Implements the [Current Time GATT service](#).

*Note:* partial implementation, no notifications or time reference info.

Copyright (c) 2019 Deomid "rojer" Ryabkov

All rights reserved

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

```
http://www.apache.org/licenses/LICENSE-2.0
```

Unless required by applicable law or agreed to in writing, software  
distributed under the License is distributed on an "AS IS" BASIS,  
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
See the License for the specific language governing permissions and  
limitations under the License.

## Overview

This library provides Mongoose OS driver for the I2C button on the [WEMOS OLED shield](#).

## Usage

```

include "mgos_lolin_button.h"
<p class="mume-header" id="include-mgos_lolin_buttonh"></p>

static void lolin_button_handler(int ev, void *ev_data, void *userdata) {
 const struct mgos_lolin_button_status *bs = (const struct mgos_lolin_button_status *) ev_data;
 const char *bn = NULL;
 switch (ev) {
 case MGOS_EV_LOLIN_BUTTON_A:
 bn = "A";
 break;
 case MGOS_EV_LOLIN_BUTTON_B:
 bn = "B";
 break;
 default:
 return;
 }
 const char *sn = NULL;
 switch (bs->state) {
 case MGOS_LOLIN_BUTTON_RELEASE:
 sn = "released";
 break;
 case MGOS_LOLIN_BUTTON_PRESS:
 sn = "pressed";
 break;
 case MGOS_LOLIN_BUTTON_DOUBLE_PRESS:
 sn = "double-pressed";
 break;
 case MGOS_LOLIN_BUTTON_LONG_PRESS:
 sn = "long-pressed";
 break;
 case MGOS_LOLIN_BUTTON_HOLD:
 sn = "held";
 break;
 }
 LOG(LL_INFO, ("Button %s %s", bn, sn));
 (void) userdata;
}
mgos_event_add_group_handler(MGOS_EV_LOLIN_BUTTON_BASE, lolin_button_handler, NULL);

```

# BH1750 ambient light sensor native library for Mongoose OS

## Overview

BH1750 ambient light sensor native C driver for Mongoose OS

## Sensor wiring

## Todo

- implement sensor sensitivity adjusting
- use timer instead of mgos\_msleep?

## Usage

```

struct sensor_bh1750* bh = sensor_bh1750_create(0x23);

float lux = sensor_bh1750_get_lux(bh, ONE_TIME_HIGH_RES_MODE_2);
printf("Light: %f lux\n", lux);

sensor_bh1750_free(bh, false);
```# A native Mongoose OS DS3231 RTC library

## Implementation
The library implements 2 structures: `struct mgos_ds3231` and a helper `struct mgos_ds3231_date_time`. Both structures are available for mJS too. The mJS support is available if the user includes the mjs library:
```

- origin: <https://github.com/mongoose-os-libs/mjs>

```

#### struct mgos_ds3231
This is used to communicate with the DS3231.
The RTC data is read into and written from using the helper `struct mgos_ds3231_date_time`
```

```

##### Example in C
``c
struct mgos_ds3231* ds=mgos_ds3231_create(addr);
time_t unixtime=time(NULL);
mgos_ds3231_write_unixtime(ds, unixtime);
mgos_ds3231_free(ds);
```

Example in mJS

```

let ds=DS3231.create(addr);
let unixtime=Timer.now();
ds.writeUnixtime(unixtime);
ds.free();
```

struct mgos_ds3231_date_time

Encapsulates the DS3231 date/time information plus the unix timestamp.

Several functions are provided to create and free the structure and to get/set different fields.

Example in C

```

struct mgos_ds3231_date_time* dt=mgos_ds3231_date_time_create();
mgos_ds3231_date_time_set_date(dt, 2016, 2, 3);
// mgos_ds3231_date_time_set_time will calculate the unixtime.
// It is VERY important to call mgos_ds3231_date_time_set_date first
mgos_ds3231_date_time_set_time(dt, 12, 34, 56);

struct mgos_ds3231* ds=mgos_ds3231_create(addr);
mgos_ds3231_write(ds, dt);
mgos_ds3231_settimeofday(ds);

mgos_ds3231_free(ds);
mgos_ds3231_date_time_free(dt);
```

Example in mJS

```

let dt=DS3231DateTime.create();
dt.setDate(2016, 2, 3);
dt.setTime(12, 34, 56);

let ds=DS3231.create(addr);
ds.write(dt);
ds.setTimeOfDay();

ds.free();
dt.free();
```

Winbond W25XXX SPI NAND Flash Driver

This is a VFS device driver for [Winbond W25 series](#) SPI NAND flash chips.

W25N01 and W25M02 (2Gb multi-die version) are supported.

BH1730 sensor support

This library provides support for the digital 16-bit ambient light sensor

BH1730.

See API documentation: for [C](#)

and [mJS](#).

RPC - Remote Procedure Calls

See [MG-RPC Guide](#)

for detailed documentation.

A Cron Mongoose OS library

Overview

This library implement cron core functionality. See also:

[crontab](#).

This library supports classic cron expressions with additional field for the seconds, plus a few extras. We'll get back to the extras later, let's start with regular cron expressions.

Classic cron expressions.

Each expression contains 6 fields:

- seconds, 0-59
- minutes, 0-59
- hours, 0-23
- day of month, 1-31
- month, 1-12 or JAN-DEC
- day of week 0-6 or SUN-SAT

Each field can contain multiple values separated with commas, and/or ranges determined by the beginning of the range, a hyphen, and the ending of the range. For example, for the day of week, it could be MON-THU,SAT

A slash can be used to specify intervals: e.g. */5 in seconds field means "every 5 seconds".

Each field can contain an asterisk * which means "any value".

Examples:

- */15 * 1-4 * * * : Run every 15 seconds from 1 to 4 hours;

- `0 */2 1-4 * * *` : Run every two minutes from 1 to 4 hours;
- `0 0 7 * * MON-FRI` : Run at 7:00 every working day;
- `0 30 23 30 * * *` : Run at 23:30 every 30th day of month.

Randomized cron expressions

It's possible to specify the time range in which a job will run certain number of times.

For example, this:

```
@random:{ "from":"10 * * * *", "to":"50 * * * *", "number":5}
```

means that the cron job will be fired in between of 10 and 50 seconds of every minute, and the total number of invocations in that 40-second time window will be approximately 5. It's not guaranteed to be exactly 5 though: might be a bit less or more.

Another example: approximately 10 invocations in between of 08:00 and 22:00 on working days:

```
@random:{ "from":"0 0 8 * * MON-FRI", "to":"0 0 22 * * MON-FRI", "number":10}
```

Note that the timeout between `from` and `to` should always be the same.

For example, this would make no sense:

```
@random:{ "from":"0 0 8 * * MON-FRI", "to":"0 0 22 * * *", "number":10}
```

Currently these cases are not checked yet, so the above expression will just result in incorrect behavior.

Sunrise/sunset

Finally, instead of specifying exact time with the first three fields (seconds, minutes, hours) one could use `@sunrise` / `@sunset`, plus optional offset like `+2h45m` or `-1.5h`.

For that to work, the device needs to have latitude/longitude and timezone configured.

Examples:

- `@sunrise * * MON-FRI` : Run at the sunrise on working days;
- `@sunset-1h30m 1 * *` : Run 1.5 hours before the sunset every 1th day of month;
- `@sunset * * *` : Run at the sunset every day

As a shortcut for the last case, when all fields are `*`, these fields can be omitted. Thus, more valid examples:

- `@sunrise`
- `@sunset`
- `@sunset+3h`

A collection of libraries for demoing Mongoose OS

Overview

This library is intended to be used for apps that demonstrate Mongoose OS capabilities. It is essentially a collection of libraries that implement a wide set of functionalities - from hardware peripherals API to cloud integrations like AWS IoT, Google IoT Core, etc.

Storage device and filesystem init table

Overview

Configures storage devices, creates and/or mounts filesystems according to configuration.

Configuration

This library defines two configuration sections: `devtab` (5 entries) and `fstab` (3 entries).

Devtab defines VFS devices to be created, fstab defines filesystems to create and/or mount.

Devtab is processed first, then fstab.

See `config_schema` section of the [manifest](#) for detailed description of settings.

Examples

- External SPI flash formatted as LFS and mounted on `/data`. Size is auto-detected.

```
libs:  
  - origin: https://github.com/mongoose-os-libs/fstab  
  - origin: https://github.com/mongoose-os-libs/vfs-dev-spi-flash  
  - origin: https://github.com/mongoose-os-libs/vfs-fs-lfs  
  
config_schema:  
  - ["spi.enable", true]  
  # Other SPI interface options go here.  
  - ["devtab.dev0.name", "spif0"]  
  - ["devtab.dev0.type", "spi_flash"]  
  - ["devtab.dev0.opts", '{"cs": 0, "freq": 10000000}']  
  - ["fstab.fs0.dev", "spif0"]  
  - ["fstab.fs0.type", "LFS"]  
  - ["fstab.fs0.opts", '{"bs": 4096}']  
  - ["fstab.fs0.path", "/data"]  
  - ["fstab.fs0.create", true]
```

- (ESP32) Additional partition on the ESP32 system flash, formatted as LFS and mounted on `/data`.

Note: All the data ESP32 partitions are automatically registered so there are no explicit devtab entries.

```

libs:
- origin: https://github.com/mongoose-os-libs/fstab
- origin: https://github.com/mongoose-os-libs/vfs-fs-lfs

build_vars:
ESP_IDF_EXTRA_PARTITION: data,data,spiffs,,256K

config_schema:
- ["fstab.fs0.dev", "data"]
- ["fstab.fs0.type", "LFS"]
- ["fstab.fs0.opts", '{"bs": 4096}']
- ["fstab.fs0.path", "/data"]
- ["fstab.fs0.create", true]

```

- External SPI flash split into two parts

First formatted for SPIFFS, the rest is not used.

```

libs:
- origin: https://github.com/mongoose-os-libs/fstab
- origin: https://github.com/mongoose-os-libs/vfs-dev-part
- origin: https://github.com/mongoose-os-libs/vfs-dev-spi-flash
- origin: https://github.com/mongoose-os-libs/vfs-fs-spiffs

config_schema:
- ["spi.enable", true]
# Other SPI interface options go here.
- ["devtab.dev0.name", "spif0"]
- ["devtab.dev0.type", "spi_flash"]
- ["devtab.dev0.opts", '{"cs": 0, "freq": 10000000}']
- ["devtab.dev1.name", "spif0p1"]
- ["devtab.dev1.type", "part"]
- ["devtab.dev1.opts", {"dev": "spif0", "offset": 0, "size": 131072}]
- ["devtab.dev2.name", "spif0p2"]
- ["devtab.dev2.type", "part"]
- ["devtab.dev2.opts", {"dev": "spif0", "offset": 131072}]
- ["fstab.fs0.dev", "spif0p1"]
- ["fstab.fs0.type", "SPIFFS"]
- ["fstab.fs0.opts", {"bs": 4096, "ps": 128, "es": 4096}]
- ["fstab.fs0.path", "/data"]
- ["fstab.fs0.create", true]

```

VFS for RAM

Location library

Overview

Provides a function to get latitude and longitude; so far it merely returns the configured values. Example usage, in the app's `mos.yml`:

```

libs:
- origin: https://github.com/mongoose-os-libs/location

config_schema:
- ["device.location.lat", 53.3242381]
- ["device.location.lon", -6.385785]

```

A blank Mongoose OS library

Overview

This blank library is intended to be used as a skeleton for developing Mongoose OS libraries.[# How to contribute](#)

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

INA219 I2C Driver

A Mongoose library for Texas Instruments' popular I2C current sensor.

Implementation details

The `INA219` is a simple I2C device that measures a voltage drop over a shunt resistor, as well as the voltage on the bus. It is able to measure voltage up to 26V (in 4mV increments) on the bus and the voltage drop over the shunt resistor in 10uV increments with a range of 320mV, and is reasonably precise.

Using a 0.1Ohm shunt resistor, the maximum current that can be measured is 3.2A.

API Description

Create an `INA219` object using `mgos_ina219_create()`, set the shunt resistor value using `mgos_ina219_set_shunt_resistance()`, then call `mgos_ina219_get_bus_voltage()` to get the bus voltage, call `mgos_ina219_get_shunt_voltage()` to get the voltage drop over the shunt resistor, and `mgos_ina219_get_current()` to get the current flowing through the circuit.

Example application

```

#include "mgos.h"
<p class="mume-header" id="include-mgosh-16"></p>

#include "mgos_config.h"
<p class="mume-header" id="include-mgos_configh-4"></p>

#include "mgos_ina219.h"
<p class="mume-header" id="include-mgos_ina219h"></p>

static void ina219_timer_cb(void *user_data) {
    struct mgos_ina219 *sensor = (struct mgos_ina219 *)user_data;
    float bus, shunt, current, res;

    if (!sensor) return;

    mgos_ina219_get_bus_voltage(sensor, &bus);
    mgos_ina219_get_shunt_resistance(sensor, &res);
    mgos_ina219_get_shunt_voltage(sensor, &shunt);
    mgos_ina219_get_current(sensor, &current);
    LOG(LL_INFO, ("Vbus=% .3f V Vshunt=% .0f uV Rshunt=% .3f Ohm IShunt=% .1f mA",
        bus, shunt*1e6, res, current*1e3));
}
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_ina219 *sensor;

    sensor = mgos_ina219_create(mgos_i2c_get_global(), mgos_sys_config_get_ina219_i2caddr());
    if (!sensor) {
        LOG(LL_ERROR, ("Could not create INA219 sensor"));
        return false;
    }

    mgos_set_timer(1000, true, ina219_timer_cb, sensor);

    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

RPC Service - Filesystem

This service provides an ability to manage device files remotely.
It is required by the `mos ls`, `mos get`, `mos put`, `mos rm` commands.
If this library is not included in the app, those commands won't work.
It is possible to call this service programmatically via serial, HTTP/RESTful,
WebSocket, MQTT or other transports
(see [RPC section](#)) or use `mos` tool.



Below is a list of exported RPC methods and arguments:

FS.List

Get device file list. Arguments: none.

Example usage:

```
mos call FS.List
[
  "mgos_ro_vars_schema.json",
  "conf0.json",
  ...
]
```

This RPC command has a shortcut: `mos ls`:

```
mos ls
"mgos_ro_vars_schema.json",
"conf0.json",
...
```

FS.ListExt

Same as `FS.List` but also returns extra file info like file sizes.

Arguments: none.

Example usage:

```
mos call FS.ListExt
[
  {
    "name": "mgos_ro_vars_schema.json",
    "size": 332
  },
  ...
]
```

This RPC command has a shortcut: `mos ls -l`:

```
mos ls -l
api_adc.js 259
api_arch_uart.js 651
...
```

FS.Get

Returns file content. Arguments:

```
{  
  "filename": "foo.txt",      // Required. Name of the file to fetch.  
  "offset": 0,                // Optional. Offset to begin with.  
  "len": 100                 // Optional. Number of bytes to return.  
                            // If this is too large, the call may fail with OOM.  
}
```

Example usage:

```
mos call FS.Get '{"filename": "init.js"}'  
{  
  "data": "bG9hZCgnYXBpX2NvbmcZpZ...",    # Base64 encoded data  
  "left": 0                                # How many bytes left in a file  
}
```

This RPC command has a shortcut: `mos get`:

```
mos get init.js  
load('api_gpio.js');  
...
```

FS.Put

Write data into file. Write is done either by overwriting an existing content, or by appending to the existing content. Arguments:

```
{  
  "filename": "foo.txt",      // Required. Name of the file to write to.  
  "append": false,            // Optional. Overwrite or append.  
  "data": "base64 text"      // Required. Data to write.  
}
```

This RPC command has a shortcut: `mos put`. It splits large files into small chunks, and calls `FS.Put` sequentially, appending content.

```
mos put /etc/passwd foo.txt
```

FS.Remove

Delete file. Arguments:

```
{  
  "filename": "foo.txt"      // Required. Name of the file to delete  
}
```

This RPC command has a shortcut: `mos rm`:

```
mos rm foo.txt
```

FS.Mkfs

Create filesystem. Arguments:

```
{
  "dev_type": "spi_flash",           // Required. Filesystem driver name.
  "dev_opts": "...",                // Required. Device-specific options.
  "fs_type": "SPIFFS",              // Required. Filesystem type.
  "fs_opts": "{\"site\": 131072}"    // Required. Filesystem-specific options.
}
```

Example usage:

```
mos call FS.Mkfs '{"dev_type": "spi_flash", "dev_opts": "{\"freq\": 2000000, \"cs\": 0}", "fs_type": "SPIFFS", "fs_opts": {}}
```

FS.Mount

Mount filesystem. Arguments:

```
{
  "path": "/foo",                  // Required. Directory name to attach to.
  "dev_type": "...",               // See FS.Mkfs above
  "dev_opts": "...",
  "fs_type": "SPIFFS",
  "fs_opts": "..."
}
```

```
mos call FS.Mount '{"path": "/mnt", "dev_type": "spi_flash", "dev_opts": "{\"freq\": 2000000, \"cs\": 0}", "fs_type": "SPIFFS", "fs_opts": {}}
```

FS.Umount

Unmout filesystem. Arguments:

```
{
  "path": "/foo"                  // Required. Mount point to detach.
}
```

```
mos call FS.Umount '{"path": "/mnt"}'
```

A VFS device that exposes a part of another device

Example

```
vfs_dev_create("big0part", "{\"dev\": \"big0\", \"offset\": 32768, \"size\": 65536}");
```

Onewire library for Mongoose OS using the ESP32's RMT device

Based on <https://github.com/nodemcu/nodemcu-firmware/blob/dev-esp32/components/platform/onewire.c>

It uses 2 RMT channels, one for RX and one for TX.

Create the onewire object like this:

```
OnewireRmt* mgos_onewire_rmt_create(int pin,int rmt_rx,int rmt_tx);
```

or

```
OnewireRmt(uint8_t pin,uint8_t rmt_rx,uint8_t rmt_tx);
```

The design of the little filesystem

A little fail-safe filesystem designed for embedded systems.



For a bit of backstory, the littlefs was developed with the goal of learning more about filesystem design by tackling the relative unsolved problem of managing a robust filesystem resilient to power loss on devices with limited RAM and ROM.

The embedded systems the littlefs is targeting are usually 32 bit microcontrollers with around 32KB of RAM and 512KB of ROM. These are often paired with SPI NOR flash chips with about 4MB of flash storage.

Flash itself is a very interesting piece of technology with quite a bit of nuance. Unlike most other forms of storage, writing to flash requires two operations: erasing and programming. The programming operation is relatively cheap, and can be very granular. For NOR flash specifically, byte-level programs are quite common. Erasing, however, requires an expensive operation that forces the state of large blocks of memory to reset in a destructive reaction that gives flash its name. The [Wikipedia entry](#) has more information if you are interested in how this works.

This leaves us with an interesting set of limitations that can be simplified to three strong requirements:

- 1. Power-loss resilient** - This is the main goal of the littlefs and the focus of this project.
Embedded systems are usually designed without a shutdown routine and a notable lack of user interface for recovery, so filesystems targeting embedded systems must be prepared to lose power at any given time.
Despite this state of things, there are very few embedded filesystems that handle power loss in a reasonable manner, and most can become corrupted if the user is unlucky enough.
- 2. Wear leveling** - Due to the destructive nature of flash, most flash chips have a limited number of erase cycles, usually in the order of around 100,000 erases per block for NOR flash. Filesystems that don't take wear into account can easily burn through blocks used to store frequently updated metadata.
Consider the [FAT filesystem](#), which stores a file allocation table (FAT) at a specific offset from the beginning of disk. Every block allocation will update this table, and after 100,000 updates, the block will likely go bad, rendering the filesystem unusable even if there are many more erase cycles available on the storage

as a whole.

3. **Bounded RAM/ROM** - Even with the design difficulties presented by the previous two limitations, we have already seen several flash filesystems developed on PCs that handle power loss just fine, such as the logging filesystems. However, these filesystems take advantage of the relatively cheap access to RAM, and use some rather... opportunistic... techniques, such as reconstructing the entire directory structure in RAM. These operations make perfect sense when the filesystem's only concern is erase cycles, but the idea is a bit silly on embedded systems.

To cater to embedded systems, the littlefs has the simple limitation of using only a bounded amount of RAM and ROM. That is, no matter what is written to the filesystem, and no matter how large the underlying storage is, the littlefs will always use the same amount of RAM and ROM. This presents a very unique challenge, and makes presumably simple operations, such as iterating through the directory tree, surprisingly difficult.

Existing designs?

There are of course, many different existing filesystem. Here is a very rough summary of the general ideas behind some of them.

Most of the existing filesystems fall into the one big category of filesystem designed in the early days of spinny magnet disks. While there is a vast amount of interesting technology and ideas in this area, the nature of spinny magnet disks encourage properties, such as grouping writes near each other, that don't make as much sense on recent storage types. For instance, on flash, write locality is not important and can actually increase wear.

One of the most popular designs for flash filesystems is called the [logging filesystem](#).

The flash filesystems [jffs](#) and [yafts](#) are good examples. In a logging filesystem, data is not stored in a data structure on disk, but instead the changes to the files are stored on disk. This has several neat advantages, such as the fact that the data is written in a cyclic log format and naturally wear levels as a side effect. And, with a bit of error detection, the entire filesystem can easily be designed to be resilient to power loss. The journaling component of most modern day filesystems is actually a reduced form of a logging filesystem. However, logging filesystems have a difficulty scaling as the size of storage increases. And most filesystems compensate by caching large parts of the filesystem in RAM, a strategy that is inappropriate for embedded systems.

Another interesting filesystem design technique is that of [copy-on-write \(COW\)](#).

A good example of this is the [btrfs](#) filesystem. COW filesystems can easily recover from corrupted blocks and have natural protection against power loss. However, if they are not designed with wear in mind, a COW filesystem could unintentionally wear down the root block where the COW data structures are synchronized.

Metadata pairs

The core piece of technology that provides the backbone for the littlefs is the concept of metadata pairs. The key idea here is that any metadata that needs to be updated atomically is stored on a pair of blocks tagged with

a revision count and checksum. Every update alternates between these two pairs, so that at any time there is always a backup containing the previous state of the metadata.

Consider a small example where each metadata pair has a revision count, a number as data, and the XOR of the block as a quick checksum. If we update the data to a value of 9, and then to a value of 5, here is what the pair of blocks may look like after each update:

```
block 1  block 2      block 1  block 2      block 1  block 2
-----|-----|-----|-----|-----|-----|
| rev: 1 | rev: 0 | | rev: 1 | rev: 2 | | rev: 3 | rev: 2 |
| data: 3 | data: 0 | -> | data: 3 | data: 9 | -> | data: 5 | data: 9 |
| xor: 2 | xor: 0 | | xor: 2 | xor: 11 | | xor: 6 | xor: 11 |
-----|-----|-----|-----|-----|
          let data = 9          let data = 5
```

After each update, we can find the most up to date value of data by looking at the revision count.

Now consider what the blocks may look like if we suddenly lose power while changing the value of data to 5:

```
block 1  block 2      block 1  block 2      block 1  block 2
-----|-----|-----|-----|-----|-----|
| rev: 1 | rev: 0 | | rev: 1 | rev: 2 | | rev: 3 | rev: 2 |
| data: 3 | data: 0 | -> | data: 3 | data: 9 | -x | data: 3 | data: 9 |
| xor: 2 | xor: 0 | | xor: 2 | xor: 11 | | xor: 2 | xor: 11 |
-----|-----|-----|-----|-----|
          let data = 9          let data = 5
                           powerloss!!!
```

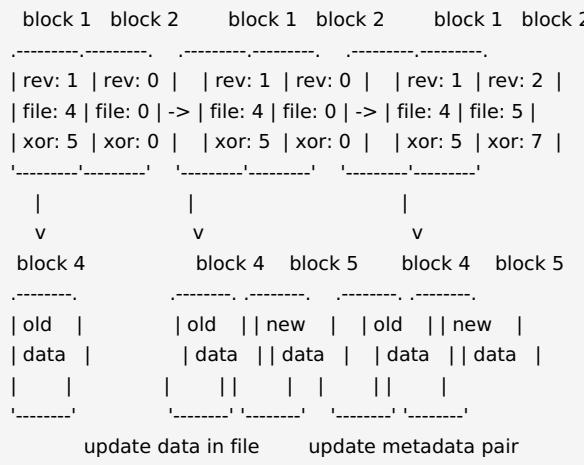
In this case, block 1 was partially written with a new revision count, but the littlefs hadn't made it to updating the value of data. However, if we check our checksum we notice that block 1 was corrupted. So we fall back to block 2 and use the value 9.

Using this concept, the littlefs is able to update metadata blocks atomically. There are a few other tweaks, such as using a 32 bit CRC and using sequence arithmetic to handle revision count overflow, but the basic concept is the same. These metadata pairs define the backbone of the littlefs, and the rest of the filesystem is built on top of these atomic updates.

Non-meta data

Now, the metadata pairs do come with some drawbacks. Most notably, each pair requires two blocks for each block of data. I'm sure users would be very unhappy if their storage was suddenly cut in half! Instead of storing everything in these metadata blocks, the littlefs uses a COW data structure for files which is in turn pointed to by a metadata block. When we update a file, we create copies of any blocks that are modified until the metadata blocks are updated with the new copy. Once the metadata block points to the new copy, we deallocate the old blocks that are no longer in use.

Here is what updating a one-block file may look like:



It doesn't matter if we lose power while writing new data to block 5, since the old data remains unmodified in block 4. This example also highlights how the atomic updates of the metadata blocks provide a synchronization barrier for the rest of the littlefs.

At this point, it may look like we are wasting an awfully large amount of space on the metadata. Just looking at that example, we are using three blocks to represent a file that fits comfortably in one! So instead of giving each file a metadata pair, we actually store the metadata for all files contained in a single directory in a single metadata block.

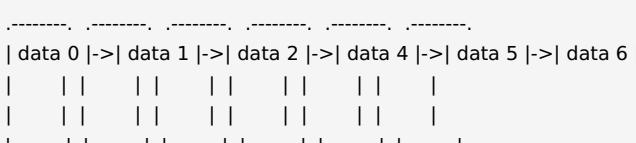
Now we could just leave files here, copying the entire file on write provides the synchronization without the duplicated memory requirements of the metadata blocks. However, we can do a bit better.

CTZ skip-lists

There are many different data structures for representing the actual files in filesystems. Of these, the littlefs uses a rather unique [COW](#) data structure that allows the filesystem to reuse unmodified parts of the file without additional metadata pairs.

First lets consider storing files in a simple linked-list. What happens when we append a block? We have to change the last block in the linked-list to point to this new block, which means we have to copy out the last block, and change the second-to-last block, and then the third-to-last, and so on until we've copied out the entire file.

Exhibit A: A linked-list



To get around this, the littlefs, at its heart, stores files backwards. Each block points to its predecessor, with the first block containing no pointers. If you think about for a while, it starts to make a bit of sense. Appending blocks just point to their predecessor and no other blocks need to be updated. If we update a block in the middle, we will need to copy out the blocks that follow, but can reuse the blocks before the modified block. Since most file operations either reset the file each write or append to files, this design avoids copying the file in the most common cases.

Exhibit B: A backwards linked-list

```
|-----|-----|-----|-----|-----|-----|  
| data 0 |<-| data 1 |<-| data 2 |<-| data 4 |<-| data 5 |<-| data 6 |  
|     ||     ||     ||     ||     ||     ||  
'-----'|-----'|-----'|-----'|-----'|-----'|
```

However, a backwards linked-list does come with a rather glaring problem. Iterating over a file *in order* has a runtime cost of $O(n^2)$. Gah! A quadratic runtime to just *read* a file? That's awful. Keep in mind reading files is usually the most common filesystem operation.

To avoid this problem, the littlefs uses a multilayered linked-list. For every n^{th} block where n is divisible by 2^x , the block contains a pointer to block $n-2^x$. So each block contains anywhere from 1 to $\log_2(n)$ pointers that skip to various sections of the preceding list. If you're familiar with data-structures, you may have recognized that this is a type of deterministic skip-list.

The name comes from the use of the [count trailing zeros \(CTZ\)](#) instruction, which allows us to calculate the power-of-two factors efficiently. For a given block n , the block contains $\text{ctz}(n)+1$ pointers.

Exhibit C: A backwards CTZ skip-list

```
|-----|-----|-----|-----|-----|-----|  
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 |<-| data 4 |<-| data 5 |  
|     ||<-||     ||<-||     ||     ||     ||  
|     ||<-||     ||<-||     ||     ||     ||  
'-----'|-----'|-----'|-----'|-----'|-----'|
```

The additional pointers allow us to navigate the data-structure on disk much more efficiently than in a singly linked-list.

Taking exhibit C for example, here is the path from data block 5 to data block 1. You can see how data block 3 was completely skipped:

```
|-----|-----|-----|-----|-----|-----|  
| data 0 | | data 1 |<-| data 2 | | data 3 | | data 4 |<-| data 5 |  
|     ||     ||     ||<-||     ||     ||     ||  
|     ||     ||     ||<-||     ||     ||     ||  
'-----'|-----'|-----'|-----'|-----'|-----'|
```

The path to data block 0 is even more quick, requiring only two jumps:

```
|-----|-----|-----|-----|-----|-----|  
| data 0 | | data 1 | | data 2 | | data 3 | | data 4 |<-| data 5 |  
|     ||     ||     ||     ||     ||     ||  
|     ||<-||     ||<-||     ||     ||     ||  
'-----'|-----'|-----'|-----'|-----'|-----'|
```

We can find the runtime complexity by looking at the path to any block from the block containing the most pointers. Every step along the path divides the search space for the block in half. This gives us a runtime of $O(\log n)$. To get to the block with the most pointers, we can perform the same steps backwards, which puts the runtime at $O(2 \log n) = O(\log n)$. The interesting part about this data structure is that this optimal path occurs naturally if we greedily choose the pointer that covers the most distance without passing our target block.

So now we have a representation of files that can be appended trivially with a runtime of $O(1)$, and can be read with a worst case runtime of $O(n \log n)$. Given that the runtime is also divided by the amount of data we can store in a block, this is pretty reasonable.

Unfortunately, the CTZ skip-list comes with a few questions that aren't straightforward to answer. What is the overhead? How do we handle more pointers than we can store in a block? How do we store the skip-list in a directory entry?

One way to find the overhead per block is to look at the data structure as multiple layers of linked-lists. Each linked-list skips twice as many blocks as the previous linked-list. Another way of looking at it is that each linked-list uses half as much storage per block as the previous linked-list. As we approach infinity, the number of pointers per block forms a geometric series. Solving this geometric series gives us an average of only 2 pointers per block.

Finding the maximum number of pointers in a block is a bit more complicated, but since our file size is limited by the integer width we use to store the size, we can solve for it. Setting the overhead of the maximum pointers equal to the block size we get the following equation. Note that a smaller block size results in more pointers, and a larger word width results in larger pointers.

where:

B = block size in bytes

w = word width in bits

Solving the equation for B gives us the minimum block size for various word widths:

32 bit CTZ skip-list = minimum block size of 104 bytes

64 bit CTZ skip-list = minimum block size of 448 bytes

Since littlefs uses a 32 bit word size, we are limited to a minimum block size of 104 bytes. This is a perfectly reasonable minimum block size, with most block sizes starting around 512 bytes. So we can avoid additional logic to avoid overflowing our block's capacity in the CTZ skip-list.

So, how do we store the skip-list in a directory entry? A naive approach would be to store a pointer to the head of the skip-list, the length of the file in bytes, the index of the head block in the skip-list, and the offset in the head block in bytes. However this is a lot of information, and we can observe that a file size maps to only one block index + offset pair. So it should be sufficient to store only the pointer and file size.

But there is one problem, calculating the block index + offset pair from a file size doesn't have an obvious implementation.

We can start by just writing down an equation. The first idea that comes to mind is to just use a for loop to sum together blocks until we reach our file size. We can write this equation as a summation:

where:

B = block size in bytes

w = word width in bits

n = block index in skip-list

N = file size in bytes

And this works quite well, but is not trivial to calculate. This equation requires O(n) to compute, which brings the entire runtime of reading a file to O($n^2 \log n$). Fortunately, the additional O(n) does not need to touch disk, so it is not completely unreasonable. But if we could solve this equation into a form that is easily computable, we can avoid a big slowdown.

Unfortunately, the summation of the CTZ instruction presents a big challenge.

How would you even begin to reason about integrating a bitwise instruction?

Fortunately, there is a powerful tool I've found useful in these situations:

The [On-Line Encyclopedia of Integer Sequences \(OEIS\)](#).

If we work out the first couple of values in our summation, we find that CTZ

maps to [A001511](#), and its partial summation maps

to [A005187](#), and surprisingly, both of these

sequences have relatively trivial equations! This leads us to a rather

unintuitive property:

where:

$\text{ctz}(x)$ = the number of trailing bits that are 0 in x

$\text{popcount}(x)$ = the number of bits that are 1 in x

It's a bit bewildering that these two seemingly unrelated bitwise instructions are related by this property. But if we start to dissect this equation we can see that it does hold. As n approaches infinity, we do end up with an average overhead of 2 pointers as we find earlier. And popcount seems to handle the error from this average as it accumulates in the CTZ skip-list.

Now we can substitute into the original equation to get a trivial equation

for a file size:

Unfortunately, we're not quite done. The popcount function is non-injective, so we can only find the file size from the block index, not the other way around. However, we can solve for an n' block index that is greater than n with an error bounded by the range of the popcount function. We can then repeatedly substitute this n' into the original equation until the error is smaller than the integer division. As it turns out, we only need to perform this substitution once. Now we directly calculate our block index:

Now that we have our block index n, we can just plug it back into the above equation to find the offset. However, we do need to rearrange the equation a bit to avoid integer overflow:

The solution involves quite a bit of math, but computers are very good at math.

Now we can solve for both the block index and offset from the file size in O(1).

Here is what it might look like to update a file stored with a CTZ skip-list:

```

        block 1  block 2
        -----.
| rev: 1 | rev: 0 |
| file: 6 | file: 0 |
| size: 4 | size: 0 |
| xor: 3 | xor: 0 |
'-----'
|
v

```

block 3 block 4 block 5 block 6

```

-----.
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 | |
|     |<-|     |--|     ||     |
|     ||     ||     ||     ||
'-----'

```

| update data in file

v

block 1 block 2

```

-----.
| rev: 1 | rev: 0 |
| file: 6 | file: 0 |
| size: 4 | size: 0 |
| xor: 3 | xor: 0 |
'-----'
|
v

```

block 3 block 4 block 5 block 6

```

-----.
| data 0 |<-| data 1 |<-| old  |<-| old  | |
|     |<-|     |--| data 2 | | data 3 |
|     ||     ||     ||     ||
'-----'

```

^ ^ ^

```

||     |   block 7   block 8   block 9   block 10
||     |   -----.
||     '---| new   |<-| new   |<-| new   |<-| new   |
|'-----| data 2 |<-| data 3 |--| data 4 | | data 5 |
'-----|     |--|     |--|     ||     |
'-----'

```

| update metadata pair

v

block 1 block 2

```

-----.
| rev: 1 | rev: 2 |
| file: 6 | file: 10|
| size: 4 | size: 6 |
| xor: 3 | xor: 14 |
'-----'
|

```

block 3 block 4 block 5 block 6

```

-----.
| data 0 |<-| data 1 |<-| old  |<-| old  | |
|     |<-|     |--| data 2 | | data 3 |
|     ||     ||     ||     ||
'-----'

```

^ ^ ^

```

||     |   block 7   block 8   block 9   block 10
||     |   -----.
||     '---| new   |<-| new   |<-| new   |<-| new   |
|'-----| data 2 |<-| data 3 |--| data 4 | | data 5 |
'-----|     |--|     |--|     ||     |
'-----'

```

Block allocation

So those two ideas provide the grounds for the filesystem. The metadata pairs give us directories, and the CTZ skip-lists give us files. But this leaves one big elephant

of a question. How do we get those blocks in the first place?

One common strategy is to store unallocated blocks in a big free list, and initially the littlefs was designed with this in mind. By storing a reference to the free list in every single metadata pair, additions to the free list could be updated atomically at the same time the replacement blocks were stored in the metadata pair. During boot, every metadata pair had to be scanned to find the most recent free list, but once the list was found the state of all free blocks becomes known.

However, this approach had several issues:

- There was a lot of nuanced logic for adding blocks to the free list without modifying the blocks, since the blocks remain active until the metadata is updated.
- The free list had to support both additions and removals in FIFO order while minimizing block erases.
- The free list had to handle the case where the file system completely ran out of blocks and may no longer be able to add blocks to the free list.
- If we used a revision count to track the most recently updated free list, metadata blocks that were left unmodified were ticking time bombs that would cause the system to go haywire if the revision count overflowed.
- Every single metadata block wasted space to store these free list references.

Actually, to simplify, this approach had one massive glaring issue: complexity.

Complexity leads to fallibility.

Fallibility leads to unmaintainability.

Unmaintainability leads to suffering.

Or at least, complexity leads to increased code size, which is a problem for embedded systems.

In the end, the littlefs adopted more of a "drop it on the floor" strategy. That is, the littlefs doesn't actually store information about which blocks are free on the storage. The littlefs already stores which files are in use, so to find a free block, the littlefs just takes all of the blocks that exist and subtract the blocks that are in use.

Of course, it's not quite that simple. Most filesystems that adopt this "drop it on the floor" strategy either rely on some properties inherent to the filesystem, such as the cyclic-buffer structure of logging filesystems, or use a bitmap or table stored in RAM to track free blocks, which scales with the size of storage and is problematic when you have limited RAM. You could iterate through every single block in storage and check it against every single block in the filesystem on every single allocation, but that would have an abhorrent runtime.

So the littlefs compromises. It doesn't store a bitmap the size of the storage, but it does store a little bit-vector that contains a fixed set lookahead for block allocations. During a block allocation, the lookahead vector is checked for any free blocks. If there are none, the lookahead region jumps forward and the entire filesystem is scanned for free blocks.

Here's what it might look like to allocate 4 blocks on a decently busy

filesystem with a 32bit lookahead and a total of
128 blocks (512Kbytes of storage if blocks are 4Kbyte):

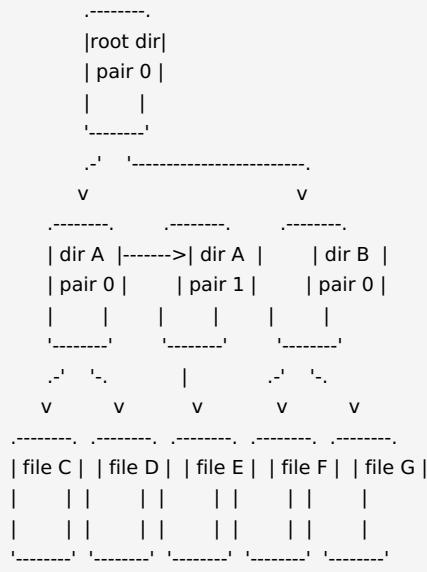
```
boot...      lookahead:  
    fs blocks: ffffff9ffffffffffffefffffff0000  
scanning...  lookahead: fffff9ff  
    fs blocks: fffff9ffffffffffffefffffff0000  
alloc = 21   lookahead: fffffdff  
    fs blocks: fffffdffffffffffffefffffff0000  
alloc = 22   lookahead: ffffffff  
    fs blocks: ffffffff0000  
scanning...  lookahead:     ffffffe  
    fs blocks: ffffffff0000  
alloc = 63   lookahead:     ffffff  
    fs blocks: ffffff0000  
scanning...  lookahead:     ffffff  
    fs blocks: ffffff0000  
scanning...  lookahead:     ffffff  
    fs blocks: ffffff0000  
scanning...  lookahead:     ffff0000  
    fs blocks: ffff0000  
alloc = 112   lookahead:         ffff8000  
    fs blocks: ffffff8000
```

While this lookahead approach still has an asymptotic runtime of $O(n^2)$ to scan all of storage, the lookahead reduces the practical runtime to a reasonable amount. Bit-vectors are surprisingly compact, given only 16 bytes, the lookahead could track 128 blocks. For a 4Mbyte flash chip with 4Kbyte blocks, the littlefs would only need 8 passes to scan the entire storage.

The real benefit of this approach is just how much it simplified the design of the littlefs. Deallocating blocks is as simple as simply forgetting they exist, and there is absolutely no concern of bugs in the deallocation code causing difficult to detect memory leaks.

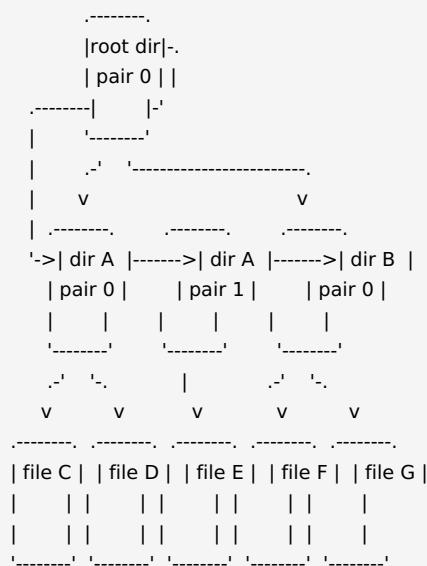
Directories

Now we just need directories to store our files. Since we already have metadata blocks that store information about files, lets just use these metadata blocks as the directories. Maybe turn the directories into linked lists of metadata blocks so it isn't limited by the number of files that fit in a single block. Add entries that represent other nested directories. Drop "." and ".." entries, cause who needs them. Dust off our hands and we now have a directory tree.



Unfortunately it turns out it's not that simple. See, iterating over a directory tree isn't actually all that easy, especially when you're trying to fit in a bounded amount of RAM, which rules out any recursive solution. And since our block allocator involves iterating over the entire filesystem tree, possibly multiple times in a single allocation, iteration needs to be efficient.

So, as a solution, the littlefs adopted a sort of threaded tree. Each directory not only contains pointers to all of its children, but also a pointer to the next directory. These pointers create a linked-list that is threaded through all of the directories in the filesystem. Since we only use this linked list to check for existence, the order doesn't actually matter. As an added plus, we can repurpose the pointer for the individual directory linked-lists and avoid using any additional space.



This threaded tree approach does come with a few tradeoffs. Now, anytime we want to manipulate the directory tree, we find ourselves having to update two pointers instead of one. For anyone familiar with creating atomic data structures this should set off a whole bunch of red flags.

But unlike the data structure guys, we can update a whole block atomically! So as long as we're really careful (and cheat a little bit), we can still manipulate the directory tree in a way that is resilient to power loss.

Consider how we might add a new directory. Since both pointers that reference it can come from the same directory, we only need a single atomic update to finagle the directory into the filesystem:

```
-----  
|root dir|.  
| pair 0 | |  
--|     |-'  
| '-----'  
|   |  
|   v  
| .-----.  
'->| dir A |  
| pair 0 |  
|   |  
'-----'  
  
| create the new directory block  
v
```

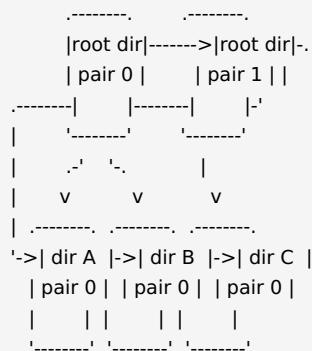
```
-----  
|root dir|.  
| pair 0 | |  
--|     |-'  
| '-----'  
|   |  
|   v  
| .-----.  
-----. '->| dir A |  
| dir B |--->| pair 0 |  
| pair 0 |   |   |  
|   |   '-----'  
'-----'
```

```
| update root to point to directory B  
v
```

```
-----  
|root dir|.  
| pair 0 | |  
-----|     |-'  
|   '-----'  
|   .-'   '-.  
|   v     v  
| .-----. .-----.  
'->| dir B |->| dir A |  
| pair 0 | | pair 0 |  
|   | |   |  
'-----' '-----'
```

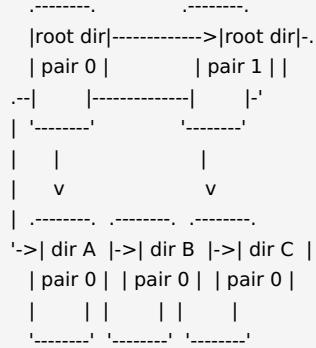
Note that even though directory B was added after directory A, we insert directory B before directory A in the linked-list because it is convenient.

Now how about removal:



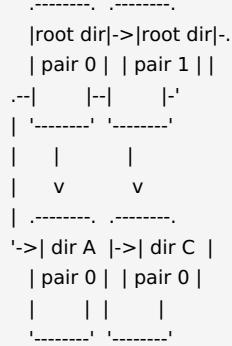
| update root to no longer contain directory B

v



| remove directory B from the linked-list

v



Wait, wait, wait, that's not atomic at all! If power is lost after removing directory B from the root, directory B is still in the linked-list. We've just created a memory leak!

And to be honest, I don't have a clever solution for this case. As a side-effect of using multiple pointers in the threaded tree, the littlefs can end up with orphan blocks that have no parents and should have been removed.

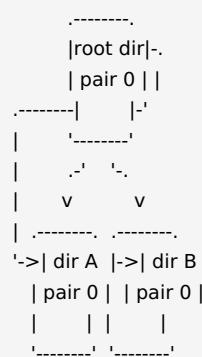
To keep these orphan blocks from becoming a problem, the littlefs has a deorphan step that simply iterates through every directory in the linked-list and checks it against every directory entry in the filesystem to see if it has a parent. The deorphan step occurs on the first block allocation after boot, so orphans should never cause the littlefs to run out of storage prematurely. Note that the deorphan step never needs to run in a read-only filesystem.

The move problem

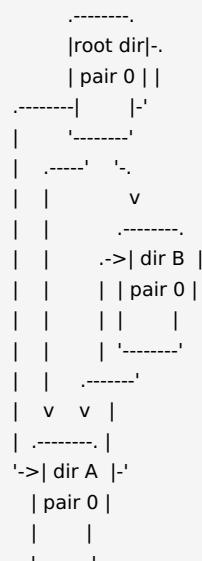
Now we have a real problem. How do we move things between directories while

remaining power resilient? Even looking at the problem from a high level, it seems impossible. We can update directory blocks atomically, but atomically updating two independent directory blocks is not an atomic operation.

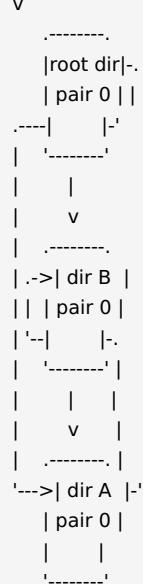
Here's the steps the filesystem may go through to move a directory:



| update directory B to point to directory A
v



| update root to no longer contain directory A
v



We can leave any orphans up to the deorphan step to collect, but that doesn't help the case where dir A has both dir B and the root dir as parents if we lose power inconveniently.

Initially, you might think this is fine. Dir A *might* end up with two parents,

but the filesystem will still work as intended. But then this raises the question of what do we do when the dir A wears out? For other directory blocks we can update the parent pointer, but for a dir with two parents we would need work out how to update both parents. And the check for multiple parents would need to be carried out for every directory, even if the directory has never been moved.

It also presents a bad user-experience, since the condition of ending up with two parents is rare, it's unlikely user-level code will be prepared. Just think about how a user would recover from a multi-parented directory. They can't just remove one directory, since remove would report the directory as "not empty".

Other atomic filesystems simple COW the entire directory tree. But this introduces a significant bit of complexity, which leads to code size, along with a surprisingly expensive runtime cost during what most users assume is a single pointer update.

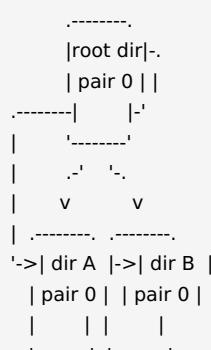
Another option is to update the directory block we're moving from to point to the destination with a sort of predicate that we have moved if the destination exists. Unfortunately, the omnipresent concern of wear could cause any of these directory entries to change blocks, and changing the entry size before a move introduces complications if it spills out of the current directory block.

So how do we go about moving a directory atomically?

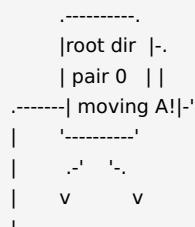
We rely on the improbableness of power loss.

Power loss during a move is certainly possible, but it's actually relatively rare. Unless a device is writing to a filesystem constantly, it's unlikely that a power loss will occur during filesystem activity. We still need to handle the condition, but runtime during a power loss takes a back seat to the runtime during normal operations.

So what littlefs does is inelegantly simple. When littlefs moves a file, it marks the file as "moving". This is stored as a single bit in the directory entry and doesn't take up much space. Then littlefs moves the directory, finishing with the complete remove of the "moving" directory entry.



| update root directory to mark directory A as moving
v



```
'->| dir A |->| dir B |
 | pair 0 | | pair 0 |
 |       | |       |
 '-----'
```

```
| update directory B to point to directory A
```

```
v
```

```
-----.
|root dir |-
| pair 0 |||
-----| moving A!|-'
| '-----'
| | .-.
| |   v
| |   -----
| |   .->| dir B | |
| |   | | pair 0 |
| |   | |   |
| |   | |   |
| |   | |   |
| |   | |   |
| |   | |   |
| |   v   v |
| |   -----|
'->| dir A |-'
| pair 0 |
|   |
'-----'
```

```
| update root to no longer contain directory A
```

```
v
```

```
-----.
|root dir|-
| pair 0 ||
-----|   |'-'
| '-----'
|   |
|   v
|   -----
| .->| dir B |
| | | pair 0 |
| | '-|   |-
| |   '-----'
| |   |
| |   v
| |   |
| |   -----
'--->| dir A |-'
| pair 0 |
|   |
'-----'
```

Now, if we run into a directory entry that has been marked as "moved", one of two things is possible. Either the directory entry exists elsewhere in the filesystem, or it doesn't. This is a $O(n)$ operation, but only occurs in the unlikely case we lost power during a move.

And we can easily fix the "moved" directory entry. Since we're already scanning the filesystem during the deorphan step, we can also check for moved entries. If we find one, we either remove the "moved" marking or remove the whole entry if it exists elsewhere in the filesystem.

Wear awareness

So now that we have all of the pieces of a filesystem, we can look at a more subtle attribute of embedded storage: The wear down of flash blocks.

The first concern for the littlefs, is that perfectly valid blocks can suddenly

become unusable. As a nice side-effect of using a COW data-structure for files, we can simply move on to a different block when a file write fails. All modifications to files are performed in copies, so we will only replace the old file when we are sure none of the new file has errors. Directories, on the other hand, need a different strategy.

The solution to directory corruption in the littlefs relies on the redundant nature of the metadata pairs. If an error is detected during a write to one of the metadata pairs, we seek out a new block to take its place. Once we find a block without errors, we iterate through the directory tree, updating any references to the corrupted metadata pair to point to the new metadata block. Just like when we remove directories, we can lose power during this operation and end up with a desynchronized metadata pair in our filesystem. And just like when we remove directories, we leave the possibility of a desynchronized metadata pair up to the deorphan step to clean up.

Here's what encountering a directory error may look like with all of the directories and directory pointers fully expanded:

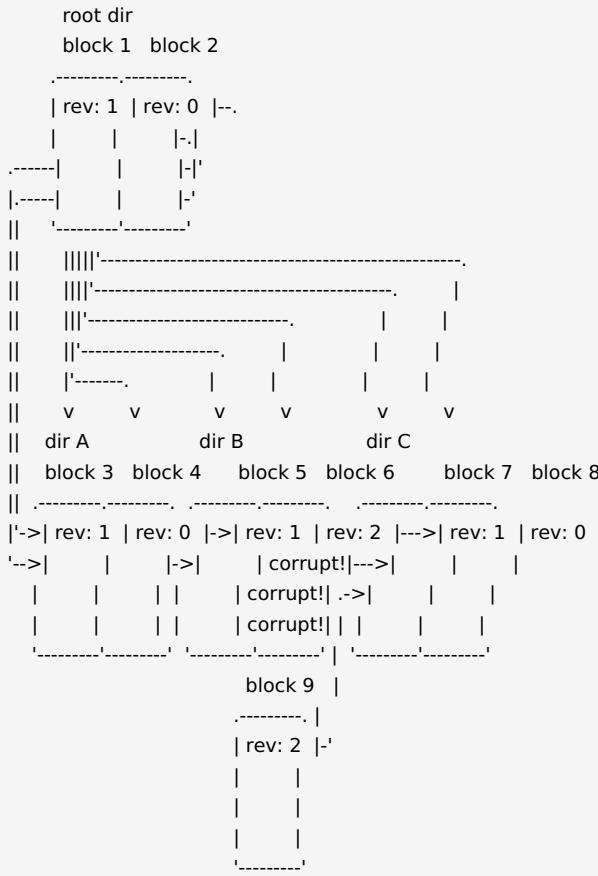
```
root dir
block 1  block 2
.-----.
| rev: 1 | rev: 0 |--.
|       |       |-'|
.----|       |       |'|
|----|       |       |'|
||   '-----'
||   ||||'-----.
||   ||||'-----.
||   ||||'-----.
||   ||'-----.
||   ||'-----.
||   ||'-----.
||   ||'-----.
||   v       v       v       v       v       v
|| dir A     dir B     dir C
|| block 3  block 4  block 5  block 6  block 7  block 8
|| .-----.
|->| rev: 1 | rev: 0 |->| rev: 1 | rev: 0 |->| rev: 1 | rev: 0 |
'-->|       |       |->|       |       |->|       |       |
|       |       |       |       |       |       |
|       |       |       |       |       |       |
|       |       |       |       |       |       |
-----.
```

```
| update directory B
```

```
v

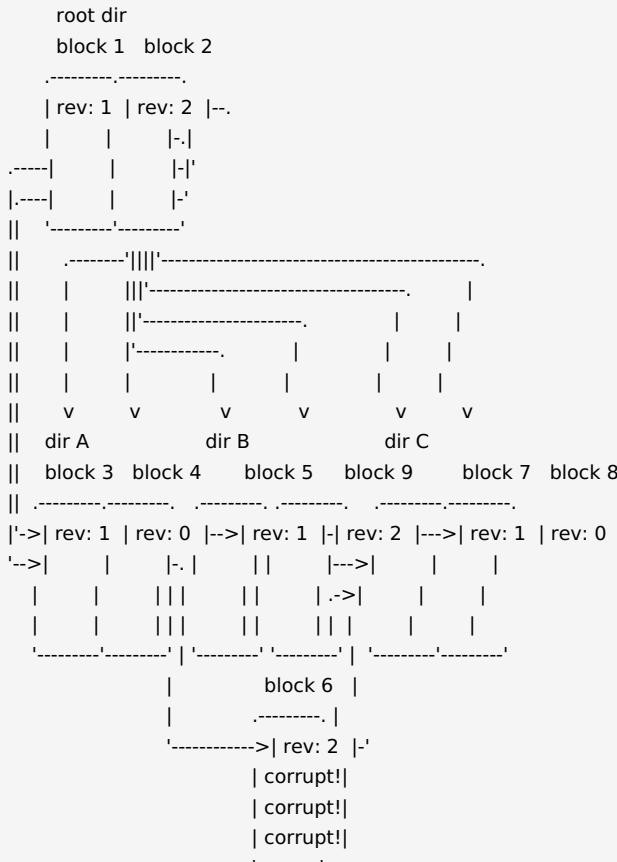
root dir
block 1  block 2
.-----.
| rev: 1 | rev: 0 |--.
|       |       |-'|
.----|       |       |'|
|----|       |       |'|
||   '-----'
||   ||||'-----.
||   ||||'-----.
||   ||||'-----.
||   ||'-----.
||   ||'-----.
||   ||'-----.
||   ||'-----.
||   v       v       v       v       v       v
|| dir A     dir B     dir C
|| block 3  block 4  block 5  block 6  block 7  block 8
|| .-----.
|->| rev: 1 | rev: 0 |->| rev: 1 | rev: 2 |->| rev: 1 | rev: 0 |
'-->|       |       |->|       | corrupt!->|       |       |
|       |       |       | corrupt!|       |       |
|       |       |       | corrupt!|       |       |
```

| oh no! corruption detected
v allocate a replacement block



| update root directory to contain block 9

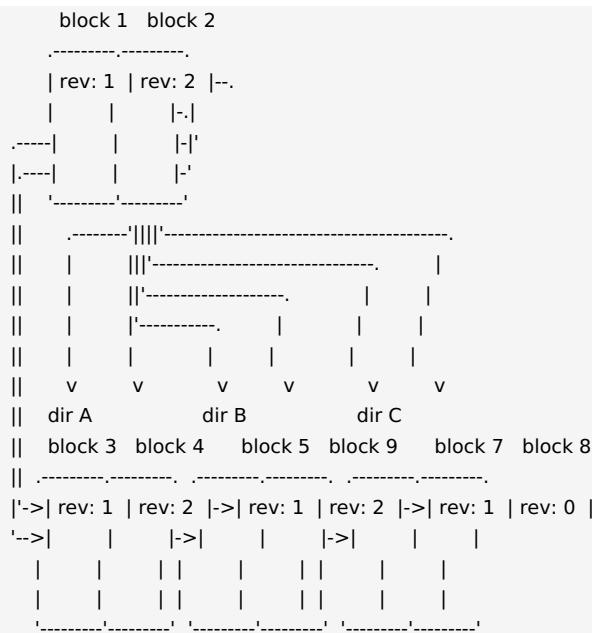
V



I remove corrupted block from linked-list

16

root dir



Also one question I've been getting is, what about the root directory? It can't move so wouldn't the filesystem die as soon as the root blocks develop errors? And you would be correct. So instead of storing the root in the first few blocks of the storage, the root is actually pointed to by the superblock. The superblock contains a few bits of static data, but outside of when the filesystem is formatted, it is only updated when the root develops errors and needs to be moved.

Wear leveling

The second concern for the littlefs is that blocks in the filesystem may wear unevenly. In this situation, a filesystem may meet an early demise where there are no more non-corrupted blocks that aren't in use. It's common to have files that were written once and left unmodified, wasting the potential erase cycles of the blocks it sits on.

Wear leveling is a term that describes distributing block writes evenly to avoid the early termination of a flash part. There are typically two levels of wear leveling:

1. Dynamic wear leveling - Wear is distributed evenly across all **dynamic** blocks. Usually this is accomplished by simply choosing the unused block with the lowest amount of wear. Note this does not solve the problem of static data.
2. Static wear leveling - Wear is distributed evenly across all **dynamic** and **static** blocks. Unmodified blocks may be evicted for new block writes. This does handle the problem of static data but may lead to wear amplification.

In littlefs's case, it's possible to use the revision count on metadata pairs to approximate the wear of a metadata block. And combined with the COW nature of files, littlefs could provide your usual implementation of dynamic wear leveling.

However, the littlefs does not. This is for a few reasons. Most notably, even if the littlefs did implement dynamic wear leveling, this would still not handle the case of write-once files, and near the end of the lifetime of a flash device, you would likely end up with uneven wear on the blocks anyways.

As a flash device reaches the end of its life, the metadata blocks will naturally be the first to go since they are updated most often. In this situation, the littlefs is designed to simply move on to another set of metadata blocks. This travelling means that at the end of a flash device's life, the filesystem will have worn the device down nearly as evenly as the usual dynamic wear leveling could. More aggressive wear leveling would come with a code-size cost for marginal benefit.

One important takeaway to note, if your storage stack uses highly sensitive storage such as NAND flash, static wear leveling is the only valid solution. In most cases you are going to be better off using a full [flash translation layer \(FTL\)](#). NAND flash already has many limitations that make it poorly suited for an embedded system: low erase cycles, very large blocks, errors that can develop even during reads, errors that can develop during writes of neighboring blocks. Managing sensitive storage such as NAND flash is out of scope for the littlefs. The littlefs does have some properties that may be beneficial on top of a FTL, such as limiting the number of writes where possible, but if you have the storage requirements that necessitate the need of NAND flash, you should have the RAM to match and just use an FTL or flash filesystem.

Summary

So, to summarize:

1. The littlefs is composed of directory blocks
2. Each directory is a linked-list of metadata pairs
3. These metadata pairs can be updated atomically by alternating which metadata block is active
4. Directory blocks contain either references to other directories or files
5. Files are represented by copy-on-write CTZ skip-lists which support O(1) append and O(n log n) reading
6. Blocks are allocated by scanning the filesystem for used blocks in a fixed-size lookahead region that is stored in a bit-vector
7. To facilitate scanning the filesystem, all directories are part of a linked-list that is threaded through the entire filesystem
8. If a block develops an error, the littlefs allocates a new block, and moves the data and references of the old block to the new.
9. Any case where an atomic operation is not possible, mistakes are resolved by a deorphan step that occurs on the first allocation after boot

That's the little filesystem. Thanks for reading!

Copyright (c) 2017, Arm Limited. All rights reserved.

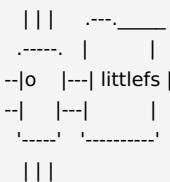
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of ARM nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The little filesystem

A little fail-safe filesystem designed for embedded systems.



Bounded RAM/ROM - The littlefs is designed to work with a limited amount of memory. Recursion is avoided and dynamic memory is limited to configurable buffers that can be provided statically.

Power-loss resilient - The littlefs is designed for systems that may have random power failures. The littlefs has strong copy-on-write guarantees and storage on disk is always kept in a valid state.

Wear leveling - Since the most common form of embedded storage is erodible flash memories, littlefs provides a form of dynamic wear leveling for systems that can not fit a full flash translation layer.

Example

Here's a simple example that updates a file named `boot_count` every time main runs. The program can be interrupted at any time without losing track of how many times it has been booted and without corrupting the filesystem:

```

# include "lfs1.h"
<p class="mume-header" id="include-lfs1h"></p>

// variables used by the filesystem
lfs1_t lfs1;
lfs1_file_t file;

// configuration of the filesystem is provided by this struct
const struct lfs1_config cfg = {
    // block device operations
    .read   = user_provided_block_device_read,
    .prog   = user_provided_block_device_prog,
    .erase   = user_provided_block_device_erase,
    .sync   = user_provided_block_device_sync,

    // block device configuration
    .read_size = 16,
    .prog_size = 16,
    .block_size = 4096,
    .block_count = 128,
    .lookahead = 128,
};

// entry point
int main(void) {
    // mount the filesystem
    int err = lfs1_mount(&lfs1, &cfg);

    // reformat if we can't mount the filesystem
    // this should only happen on the first boot
    if (err) {
        lfs1_format(&lfs1, &cfg);
        lfs1_mount(&lfs1, &cfg);
    }

    // read current count
    uint32_t boot_count = 0;
    lfs1_file_open(&lfs1, &file, "boot_count", LFS1_O_RDWR | LFS1_O_CREAT);
    lfs1_file_read(&lfs1, &file, &boot_count, sizeof(boot_count));

    // update boot count
    boot_count += 1;
    lfs1_file_rewind(&lfs1, &file);
    lfs1_file_write(&lfs1, &file, &boot_count, sizeof(boot_count));

    // remember the storage is not updated until the file is closed successfully
    lfs1_file_close(&lfs1, &file);

    // release any resources we were using
    lfs1_unmount(&lfs1);

    // print the boot count
    printf("boot_count: %d\n", boot_count);
}

```

Usage

Detailed documentation (or at least as much detail as is currently available) can be found in the comments in [lfs1.h](#).

As you may have noticed, littlefs takes in a configuration structure that defines how the filesystem operates. The configuration struct provides the filesystem with the block device operations and dimensions, tweakable parameters that tradeoff memory usage for performance, and optional static buffers if the user wants to avoid dynamic memory.

The state of the littlefs is stored in the `lfs1_t` type which is left up to the user to allocate, allowing multiple filesystems to be in use simultaneously. With the `lfs1_t` and configuration struct, a user can format a block device or mount the filesystem.

Once mounted, the littlefs provides a full set of POSIX-like file and directory functions, with the deviation that the allocation of filesystem structures must be provided by the user.

All POSIX operations, such as remove and rename, are atomic, even in event of power-loss. Additionally, no file updates are actually committed to the filesystem until sync or close is called on the file.

Other notes

All littlefs calls have the potential to return a negative error code. The errors can be either one of those found in the enum `lfs1_error` in [lfs1.h](#), or an error returned by the user's block device operations.

In the configuration struct, the `prog` and `erase` function provided by the user may return a `LFS1_ERR_CORRUPT` error if the implementation already can detect corrupt blocks. However, the wear leveling does not depend on the return code of these functions, instead all data is read back and checked for integrity.

If your storage caches writes, make sure that the provided `sync` function flushes all the data to memory and ensures that the next read fetches the data from memory, otherwise data integrity can not be guaranteed. If the `write` function does not perform caching, and therefore each `read` or `write` call hits the memory, the `sync` function can simply return 0.

Reference material

[DESIGN.md](#) - [DESIGN.md](#) contains a fully detailed dive into how littlefs actually works. I would encourage you to read it since the solutions and tradeoffs at work here are quite interesting.

[SPEC.md](#) - [SPEC.md](#) contains the on-disk specification of littlefs with all the nitty-gritty details. Can be useful for developing tooling.

Testing

The littlefs comes with a test suite designed to run on a PC using the [emulated block device](#) found in the emubd directory.

The tests assume a Linux environment and can be started with make:

```
make test
```

License

The littlefs is provided under the [BSD-3-Clause](#) license. See [LICENSE.md](#) for more information. Contributions to this project are accepted under the same license.

Individual files contain the following tag instead of the full license text.

This enables machine processing of license information based on the SPDX License Identifiers that are here available: <http://spdx.org/licenses/>

Related projects

Mbed OS -

The easiest way to get started with littlefs is to jump into [Mbed](#), which already has block device drivers for most forms of embedded storage. The littlefs is available in Mbed OS as the [LittleFileSystem](#) class.

[littlefs-fuse](#) - A FUSE

wrapper for littlefs. The project allows you to mount littlefs directly on a Linux machine. Can be useful for debugging littlefs if you have an SD card handy.

[littlefs-js](#) - A javascript wrapper for littlefs. I'm not sure why you would want this, but it is handy for demos.

You can see it in action [here](#).

[mk-lfs](#) -

A command line tool built by the [Lua RTOS](#) guys for making littlefs images from a host PC. Supports Windows, Mac OS, and Linux.

[SPIFFS](#) - Another excellent embedded filesystem for NOR flash. As a more traditional logging filesystem with full static wear-leveling, SPIFFS will likely outperform littlefs on small memories such as the internal flash on microcontrollers.

[Dhara](#) - An interesting NAND flash translation layer designed for small MCUs. It offers static wear-leveling and power-resilience with only a fixed $O(|address|)$ pointer structure stored on each block and in RAM.

The little filesystem technical specification

This is the technical specification of the little filesystem. This document covers the technical details of how the littlefs is stored on disk for introspection and tooling development. This document assumes you are familiar with the design of the littlefs, for more info on how littlefs works check out [DESIGN.md](#).

```
  |||  .--._____
  .----. |   |
--|o  |---| littlefs |
--|  |---|   |
'----' '-----'
  |||
```

Some important details

- The littlefs is a block-based filesystem. This is, the disk is divided into an array of evenly sized blocks that are used as the logical unit of storage

in littlefs. Block pointers are stored in 32 bits.

- There is no explicit free-list stored on disk, the littlefs only knows what is in use in the filesystem.
- The littlefs uses the value of 0xffffffff to represent a null block-pointer.
- All values in littlefs are stored in little-endian byte order.

Directories / Metadata pairs

Metadata pairs form the backbone of the littlefs and provide a system for atomic updates. Even the superblock is stored in a metadata pair.

As their name suggests, a metadata pair is stored in two blocks, with one block acting as a redundant backup in case the other is corrupted. These two blocks could be anywhere in the disk and may not be next to each other, so any pointers to directory pairs need to be stored as two block pointers.

Here's the layout of metadata blocks on disk:

offset	size	description
0x00	32 bits	revision count
0x04	32 bits	dir size
0x08	64 bits	tail pointer
0x10	size-16 bytes	dir entries
0x00+s	32 bits	CRC

Revision count - Incremented every update, only the uncorrupted metadata-block with the most recent revision count contains the valid metadata. Comparison between revision counts must use sequence comparison since the revision counts may overflow.

Dir size - Size in bytes of the contents in the current metadata block, including the metadata-pair metadata. Additionally, the highest bit of the dir size may be set to indicate that the directory's contents continue on the next metadata-pair pointed to by the tail pointer.

Tail pointer - Pointer to the next metadata-pair in the filesystem.

A null pair-pointer (0xffffffff, 0xffffffff) indicates the end of the list. If the highest bit in the dir size is set, this points to the next metadata-pair in the current directory, otherwise it points to an arbitrary metadata-pair. Starting with the superblock, the tail-pointers form a linked-list containing all metadata-pairs in the filesystem.

CRC - 32 bit CRC used to detect corruption from power-lost, from block end-of-life, or just from noise on the storage bus. The CRC is appended to the end of each metadata-block. The littlefs uses the standard CRC-32, which uses a polynomial of 0x04c11db7, initialized with 0xffffffff.

Here's an example of a simple directory stored on disk:

```
(32 bits) revision count = 10          (0x0000000a)
(32 bits) dir size      = 154 bytes, end of dir (0x0000009a)
(64 bits) tail pointer  = 37, 36       (0x00000025, 0x00000024)
(32 bits) CRC           = 0xc86e3106
```

```
00000000: 0a 00 00 00 9a 00 00 00 25 00 00 00 24 00 00 00 .....%....$...
00000010: 22 08 00 03 05 00 00 00 04 00 00 00 74 65 61 22 ".....tea"
00000020: 08 00 06 07 00 00 00 06 00 00 00 63 6f 66 66 65 .....coffe
00000030: 65 22 08 00 04 09 00 00 00 08 00 00 00 73 6f 64 e".....sod
00000040: 61 22 08 00 05 1d 00 00 00 1c 00 00 00 6d 69 6c a".....mil
00000050: 6b 31 22 08 00 05 1f 00 00 00 1e 00 00 00 6d 69 k1".....mi
00000060: 6c 6b 32 22 08 00 05 21 00 00 00 20 00 00 00 6d lk2"!...!....m
00000070: 69 6c 6b 33 22 08 00 05 23 00 00 00 22 00 00 00 ilk3"!#..."...
00000080: 6d 69 6c 6b 34 22 08 00 05 25 00 00 00 24 00 00 milk4"!%...$..
00000090: 00 6d 69 6c 6b 35 06 31 6e c8           .milk5.1n.
```

A note about the tail pointer linked-list: Normally, this linked-list is threaded through the entire filesystem. However, after power-loss this linked-list may become out of sync with the rest of the filesystem.

- The linked-list may contain a directory that has actually been removed
- The linked-list may contain a metadata pair that has not been updated after a block in the pair has gone bad.

The threaded linked-list must be checked for these errors before it can be used reliably. Fortunately, the threaded linked-list can simply be ignored if littlefs is mounted read-only.

Entries

Each metadata block contains a series of entries that follow a standard layout. An entry contains the type of the entry, along with a section for entry-specific data, attributes, and a name.

Here's the layout of entries on disk:

offset	size	description
0x0	8 bits	entry type
0x1	8 bits	entry length
0x2	8 bits	attribute length
0x3	8 bits	name length
0x4	entry length bytes	entry-specific data
0x4+e	attribute length bytes	system-specific attributes
0x4+e+a	name length bytes	entry name

Entry type - Type of the entry, currently this is limited to the following:

- 0x11 - file entry
- 0x22 - directory entry
- 0x2e - superblock entry

Additionally, the type is broken into two 4 bit nibbles, with the upper nibble specifying the type's data structure used when scanning the filesystem. The lower nibble clarifies the type further when multiple entries share the same data structure.

The highest bit is reserved for marking the entry as "moved". If an entry is marked as "moved", the entry may also exist somewhere else in the filesystem. If the entry exists elsewhere, this entry must be treated as though it does not exist.

Entry length - Length in bytes of the entry-specific data. This does not include the entry type size, attributes, or name. The full size in bytes of the entry is $4 + \text{entry length} + \text{attribute length} + \text{name length}$.

Attribute length - Length of system-specific attributes in bytes. Since attributes are system specific, there is not much guarantee on the values in this section, and systems are expected to work even when it is empty. See the [attributes](#) section for more details.

Name length - Length of the entry name. Entry names are stored as UTF8, although most systems will probably only support ASCII. Entry names can not contain '/' and can not be '.' or '..' as these are a part of the syntax of filesystem paths.

Here's an example of a simple entry stored on disk:

```
(8 bits) entry type      = file    (0x11)
(8 bits) entry length    = 8 bytes (0x08)
(8 bits) attribute length = 0 bytes (0x00)
(8 bits) name length     = 12 bytes (0x0c)
(8 bytes) entry data      = 05 00 00 00 20 00 00 00
(12 bytes) entry name     = smallavocado

00000000: 11 08 00 0c 05 00 00 00 20 00 00 00 73 6d 61 6c ..... .smal
00000010: 6c 61 76 61 63 61 64 6f           lavacado
```

Superblock

The superblock is the anchor for the littlefs. The superblock is stored as a metadata pair containing a single superblock entry. It is through the superblock that littlefs can access the rest of the filesystem.

The superblock can always be found in blocks 0 and 1, however fetching the superblock requires knowing the block size. The block size can be guessed by searching the beginning of disk for the string "littlefs", although currently the filesystems relies on the user providing the correct block size.

The superblock is the most valuable block in the filesystem. It is updated very rarely, only during format or when the root directory must be moved. It is encouraged to always write out both superblock pairs even though it is not required.

Here's the layout of the superblock entry:

offset	size	description
0x00	8 bits	entry type (0x2e for superblock entry)
0x01	8 bits	entry length (20 bytes)
0x02	8 bits	attribute length
0x03	8 bits	name length (8 bytes)
0x04	64 bits	root directory
0x0c	32 bits	block size

offset	size	block size
0x10	32 bits	block count
0x14	32 bits	version
0x18	attribute length bytes	system-specific attributes
0x18+a	8 bytes	magic string ("littlefs")

Root directory - Pointer to the root directory's metadata pair.

Block size - Size of the logical block size used by the filesystem.

Block count - Number of blocks in the filesystem.

Version - The littlefs version encoded as a 32 bit value. The upper 16 bits encodes the major version, which is incremented when a breaking-change is introduced in the filesystem specification. The lower 16 bits encodes the minor version, which is incremented when a backwards-compatible change is introduced. Non-standard Attribute changes do not change the version. This specification describes version 1.1 (0x00010001), which is the first version of littlefs.

Magic string - The magic string "littlefs" takes the place of an entry name.

Here's an example of a complete superblock:

```
(32 bits) revision count = 3          (0x00000003)
(32 bits) dir size     = 52 bytes, end of dir (0x00000034)
(64 bits) tail pointer = 3, 2         (0x00000003, 0x00000002)
(8 bits) entry type   = superblock   (0x2e)
(8 bits) entry length  = 20 bytes    (0x14)
(8 bits) attribute length = 0 bytes  (0x00)
(8 bits) name length   = 8 bytes    (0x08)
(64 bits) root directory = 3, 2       (0x00000003, 0x00000002)
(32 bits) block size    = 512 bytes   (0x00000200)
(32 bits) block count   = 1024 blocks (0x00000400)
(32 bits) version       = 1.1        (0x00010001)
(8 bytes) magic string  = littlefs
(32 bits) CRC           = 0xc50b74fa
```

```
00000000: 03 00 00 00 34 00 00 00 03 00 00 00 02 00 00 00 ....4.....
00000010: 2e 14 00 08 03 00 00 00 02 00 00 00 00 02 00 00 .....
00000020: 00 04 00 00 01 00 01 00 6c 69 74 74 6c 65 66 73 .....littlefs
00000030: fa 74 0b c5               .t..
```

Directory entries

Directories are stored in entries with a pointer to the first metadata pair in the directory. Keep in mind that a directory may be composed of multiple metadata pairs connected by the tail pointer when the highest bit in the dir size is set.

Here's the layout of a directory entry:

offset	size	description
0x0	8 bits	entry type (0x22 for directory entries)
0x1	8 bits	entry length (8 bytes)
0x2	8 bits	attribute length

0x3	8 bits	name length
0x4	64 bits	directory pointer
0xc	attribute length bytes	system-specific attributes
0xc+a	name length bytes	directory name

Directory pointer - Pointer to the first metadata pair in the directory.

Here's an example of a directory entry:

```
(8 bits) entry type      = directory (0x22)
(8 bits) entry length    = 8 bytes  (0x08)
(8 bits) attribute length = 0 bytes (0x00)
(8 bits) name length     = 3 bytes (0x03)
(64 bits) directory pointer = 5, 4   (0x00000005, 0x00000004)
(3 bytes) name          = tea

00000000: 22 08 00 03 05 00 00 00 04 00 00 00 74 65 61  ".....tea
```

File entries

Files are stored in entries with a pointer to the head of the file and the size of the file. This is enough information to determine the state of the CTZ skip-list that is being referenced.

How files are actually stored on disk is a bit complicated. The full explanation of CTZ skip-lists can be found in [DESIGN.md](#).

A terribly quick summary: For every nth block where n is divisible by 2^x , the block contains a pointer to block $n-2^x$. These pointers are stored in increasing order of x in each block of the file preceding the data in the block.

The maximum number of pointers in a block is bounded by the maximum file size divided by the block size. With 32 bits for file size, this results in a minimum block size of 104 bytes.

Here's the layout of a file entry:

offset	size	description
0x0	8 bits	entry type (0x11 for file entries)
0x1	8 bits	entry length (8 bytes)
0x2	8 bits	attribute length
0x3	8 bits	name length
0x4	32 bits	file head
0x8	32 bits	file size
0xc	attribute length bytes	system-specific attributes
0xc+a	name length bytes	directory name

File head - Pointer to the block that is the head of the file's CTZ skip-list.

File size - Size of file in bytes.

Here's an example of a file entry:

```
(8 bits) entry type      = file    (0x11)
(8 bits) entry length    = 8 bytes (0x08)
(8 bits) attribute length = 0 bytes (0x00)
(8 bits) name length     = 12 bytes (0x03)
(32 bits) file head      = 543     (0x00000021f)
(32 bits) file size       = 256 KB  (0x00040000)
(12 bytes) name          = largeavacado

00000000: 11 08 00 0c 1f 02 00 00 00 00 04 00 6c 61 72 67 .....larg
00000010: 65 61 76 61 63 61 64 6f           eavacado
```

Entry attributes

Each dir entry can have up to 256 bytes of system-specific attributes. Since these attributes are system-specific, they may not be portable between different systems. For this reason, all attributes must be optional. A minimal littlefs driver must be able to get away with supporting no attributes at all.

For some level of portability, littlefs has a simple scheme for attributes. Each attribute is prefixes with an 8-bit type that indicates what the attribute is. The length of attributes may also be determined from this type. Attributes in an entry should be sorted based on portability, since attribute parsing will end when it hits the first attribute it does not understand.

Each system should choose a 4-bit value to prefix all attribute types with to avoid conflicts with other systems. Additionally, littlefs drivers that support attributes should provide a "ignore attributes" flag to users in case attribute conflicts do occur.

Attribute types prefixes with 0x0 and 0xf are currently reserved for future standard attributes. Standard attributes will be added to this document in that case.

Here's an example of non-standard time attribute:

```
(8 bits) attribute type = time    (0xc1)
(72 bits) time in seconds = 1506286115 (0x0059c81a23)

00000000: c1 23 1a c8 59 00           .#..Y.
```

Here's an example of non-standard permissions attribute:

```
(8 bits) attribute type = permissions (0xc2)
(16 bits) permission bits = rw-rw-r-- (0x01b4)

00000000: c2 b4 01           ...
```

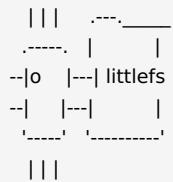
Here's what a dir entry may look like with these attributes:

```
(8 bits) entry type      = file      (0x11)
(8 bits) entry length    = 8 bytes    (0x08)
(8 bits) attribute length = 9 bytes    (0x09)
(8 bits) name length     = 12 bytes   (0x0c)
(8 bytes) entry data     = 05 00 00 00 20 00 00 00
(8 bits) attribute type  = time      (0xc1)
(72 bits) time in seconds = 1506286115 (0x0059c81a23)
(8 bits) attribute type  = permissions (0xc2)
(16 bits) permission bits = rw-rw-r-- (0x01b4)
(12 bytes) entry name    = smallavacado
```

```
00000000: 11 08 09 0c 05 00 00 00 20 00 00 00 c1 23 1a c8 ..... ....#..
00000010: 59 00 c2 b4 01 73 6d 61 6c 61 76 61 63 61 64 Y....smallavacad
00000020: 6f          o
```

The design of littlefs

A little fail-safe filesystem designed for microcontrollers.



littlefs was originally built as an experiment to learn about filesystem design in the context of microcontrollers. The question was: How would you build a filesystem that is resilient to power-loss and flash wear without using unbounded memory?

This document covers the high-level design of littlefs, how it is different than other filesystems, and the design decisions that got us here. For the low-level details covering every bit on disk, check out [SPEC.md](#).

The problem

The embedded systems littlefs targets are usually 32-bit microcontrollers with around 32 KiB of RAM and 512 KiB of ROM. These are often paired with SPI NOR flash chips with about 4 MiB of flash storage. These devices are too small for Linux and most existing filesystems, requiring code written specifically with size in mind.

Flash itself is an interesting piece of technology with its own quirks and nuance. Unlike other forms of storage, writing to flash requires two operations: erasing and programming. Programming (setting bits to 0) is relatively cheap and can be very granular. Erasing however (setting bits to 1), requires an expensive and destructive operation which gives flash its name. [Wikipedia](#) has more information on how exactly flash works.

To make the situation more annoying, it's very common for these embedded systems to lose power at any time. Usually, microcontroller code is simple and reactive, with no concept of a shutdown routine. This presents a big challenge for persistent storage, where an unlucky power loss can corrupt the storage and leave a device unrecoverable.

This leaves us with three major requirements for an embedded filesystem.

1. **Power-loss resilience** - On these systems, power can be lost at any time.

If a power loss corrupts any persistent data structures, this can cause the device to become unrecoverable. An embedded filesystem must be designed to recover from a power loss during any write operation.

2. **Wear leveling** - Writing to flash is destructive. If a filesystem repeatedly writes to the same block, eventually that block will wear out. Filesystems that don't take wear into account can easily burn through blocks used to store frequently updated metadata and cause a device's early death.

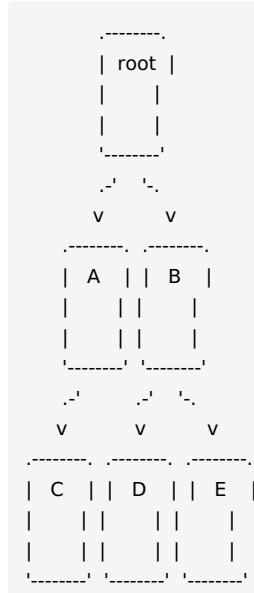
3. **Bounded RAM/ROM** - If the above requirements weren't enough, these systems also have very limited amounts of memory. This prevents many existing filesystem designs, which can lean on relatively large amounts of RAM to temporarily store filesystem metadata.

For ROM, this means we need to keep our design simple and reuse code paths were possible. For RAM we have a stronger requirement, all RAM usage is bounded. This means RAM usage does not grow as the filesystem changes in size or number of files. This creates a unique challenge as even presumably simple operations, such as traversing the filesystem, become surprisingly difficult.

Existing designs?

So, what's already out there? There are, of course, many different filesystems, however they often share and borrow feature from each other. If we look at power-loss resilience and wear leveling, we can narrow these down to a handful of designs.

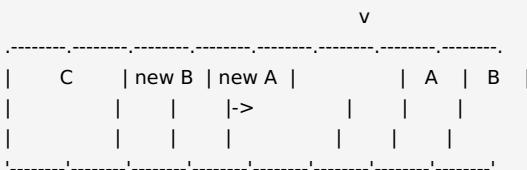
1. First we have the non-resilient, block based filesystems, such as [FAT](#) and [ext2](#). These are the earliest filesystem designs and often the most simple. Here storage is divided into blocks, with each file being stored in a collection of blocks. Without modifications, these filesystems are not power-loss resilient, so updating a file is as simple as rewriting the blocks in place.



Because of their simplicity, these filesystems are usually both the fastest and smallest. However the lack of power resilience is not great, and the binding relationship of storage location and data removes the filesystem's ability to manage wear.

2. In a completely different direction, we have logging filesystems, such as [JFFS](#), [YAFFS](#), and [SPIFFS](#), storage location is not bound to a piece of data, instead the entire storage is used for a circular log which is

appended with every change made to the filesystem. Writing appends new changes, while reading requires traversing the log to reconstruct a file. Some logging filesystems cache files to avoid the read cost, but this comes at a tradeoff of RAM.



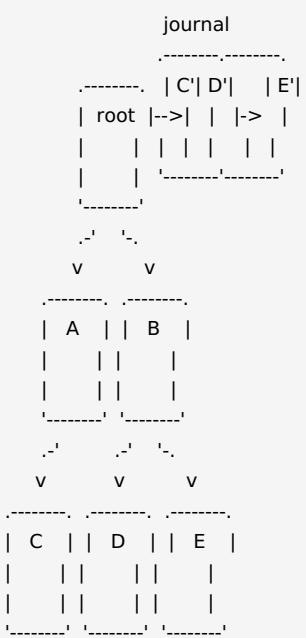
Logging filesystems are beautifully elegant. With a checksum, we can easily detect power-loss and fall back to the previous state by ignoring failed appends. And if that wasn't good enough, their cyclic nature means that logging filesystems distribute wear across storage perfectly.

The main downside is performance. If we look at garbage collection, the process of cleaning up outdated data from the end of the log, I've yet to see a pure logging filesystem that does not have one of these two costs:

1. $O(n^2)$ runtime
2. $O(n)$ RAM

SPIFFS is a very interesting case here, as it uses the fact that repeated programs to NOR flash is both atomic and masking. This is a very neat solution, however it limits the type of storage you can support.

3. Perhaps the most common type of filesystem, a journaling filesystem is the offspring that happens when you mate a block based filesystem with a logging filesystem. [ext4](#) and [NTFS](#) are good examples. Here, we take a normal block based filesystem and add a bounded log where we note every change before it occurs.

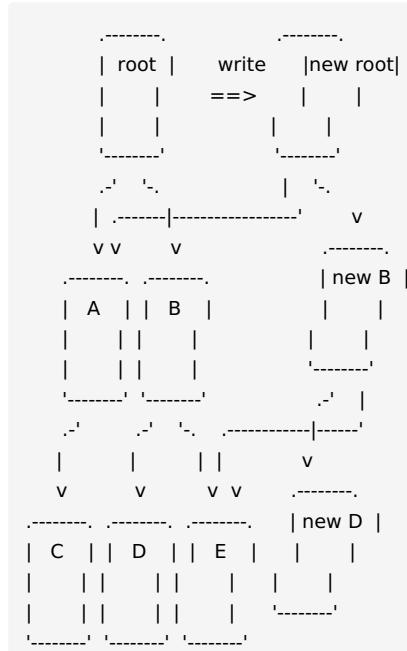


This sort of filesystem takes the best from both worlds. Performance can be as fast as a block based filesystem (though updating the journal does have a small cost), and atomic updates to the journal allow the filesystem to recover in the event of a power loss.

Unfortunately, journaling filesystems have a couple of problems. They are fairly complex, since there are effectively two filesystems running in parallel, which comes with a code size cost. They also offer no protection against wear because of the strong relationship between storage location and data.

4. Last but not least we have copy-on-write (COW) filesystems, such as

[btrfs](#) and [ZFS](#). These are very similar to other block based filesystems, but instead of updating block *in place*, all updates are performed by creating a copy with the changes and replacing any references to the old block with our new block. This recursively pushes all of our problems upwards until we reach the root of our filesystem, which is often stored in a very small log.



COW filesystems are interesting. They offer very similar performance to block based filesystems while managing to pull off atomic updates without storing data changes directly in a log. They even disassociate the storage location of data, which creates an opportunity for wear leveling.

Well, almost. The unbounded upwards movement of updates causes some problems. Because updates to a COW filesystem don't stop until they've reached the root, an update can cascade into a larger set of writes than would be needed for the original data. On top of this, the upward motion focuses these writes into the block, which can wear out much earlier than the rest of the filesystem.

littlefs

So what does littlefs do?

If we look at existing filesystems, there are two interesting design patterns that stand out, but each have their own set of problems. Logging, which provides independent atomicity, has poor runtime performance. And COW data structures, which perform well, push the atomicity problem upwards.

Can we work around these limitations?

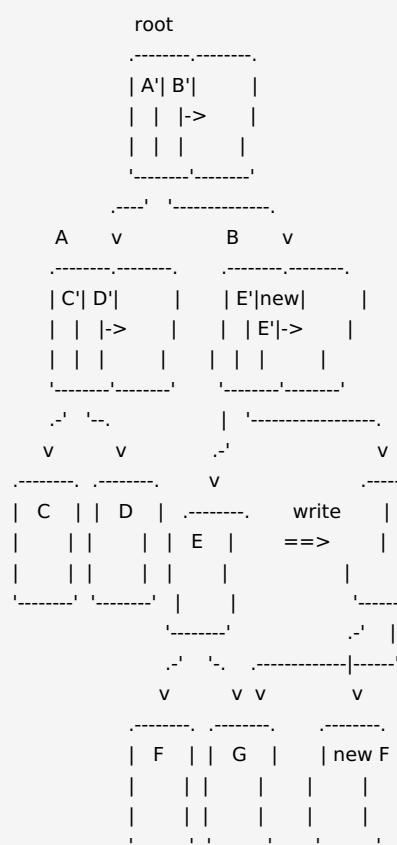
Consider logging. It has either a $O(n^2)$ runtime or $O(n)$ RAM cost. We can't avoid these costs, *but* if we put an upper bound on the size we can at least prevent the theoretical cost from becoming a problem. This relies on the super secret computer science hack where you can pretend any algorithmic complexity is $O(1)$ by bounding the input.

In the case of COW data structures, we can try twisting the definition a bit. Let's say that our COW structure doesn't copy after a single write, but instead copies after n writes. This doesn't change most COW properties (assuming you can write atomically!), but what it does do is prevent the upward motion of wear. This sort of copy-on-bounded-writes (COBW) still focuses wear, but at

each level we divide the propagation of wear by n . With a sufficiently large n (> branching factor) wear propagation is no longer a problem.

See where this is going? Separate, logging and COW are imperfect solutions and have weaknesses that limit their usefulness. But if we merge the two they can mutually solve each other's limitations.

This is the idea behind littlefs. At the sub-block level, littlefs is built out of small, two block logs that provide atomic updates to metadata anywhere on the filesystem. At the super-block level, littlefs is a CObW tree of blocks that can be evicted on demand.



There are still some minor issues. Small logs can be expensive in terms of storage, in the worst case a small log costs 4x the size of the original data. CObW structures require an efficient block allocator since allocation occurs every n writes. And there is still the challenge of keeping the RAM usage constant.

Metadata pairs

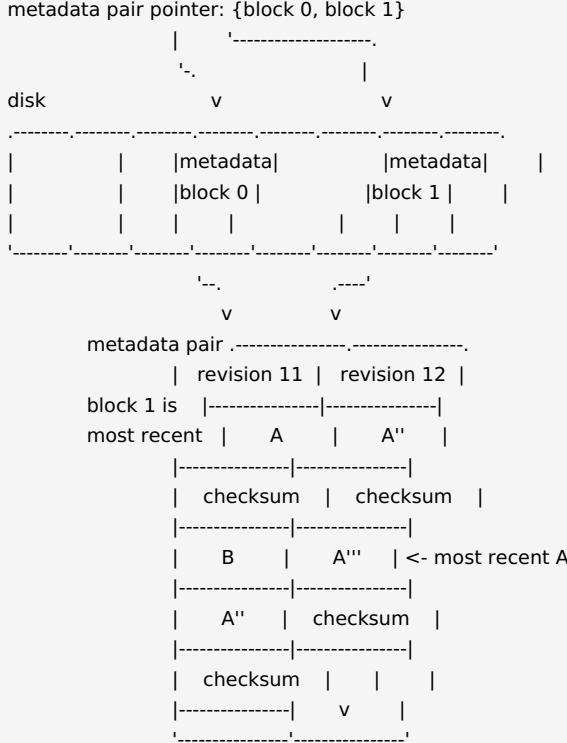
Metadata pairs are the backbone of littlefs. These are small, two block logs that allow atomic updates anywhere in the filesystem.

Why two blocks? Well, logs work by appending entries to a circular buffer stored on disk. But remember that flash has limited write granularity. We can incrementally program new data onto erased blocks, but we need to erase a full block at a time. This means that in order for our circular buffer to work, we need more than one block.

We could make our logs larger than two blocks, but the next challenge is how do we store references to these logs? Because the blocks themselves are erased during writes, using a data structure to track these blocks is complicated. The simple solution here is to store a two block addresses for every metadata

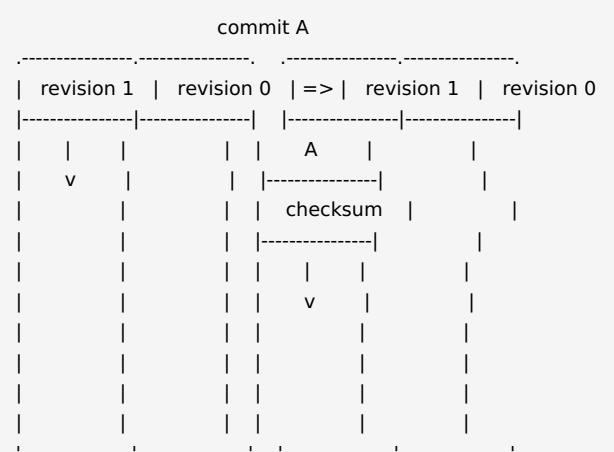
pair. This has the added advantage that we can change out blocks in the metadata pair independently, and we don't reduce our block granularity for other operations.

In order to determine which metadata block is the most recent, we store a revision count that we compare using [sequence arithmetic](#) (very handy for avoiding problems with integer overflow). Conveniently, this revision count also gives us a rough idea of how many erases have occurred on the block.



So how do we atomically update our metadata pairs? Atomicity (a type of power-loss resilience) requires two parts: redundancy and error detection. Error detection can be provided with a checksum, and in littlefs's case we use a 32-bit [CRC](#). Maintaining redundancy, on the other hand, requires multiple stages.

1. If our block is not full and the program size is small enough to let us append more entries, we can simply append the entries to the log. Because we don't overwrite the original entries (remember rewriting flash requires an erase), we still have the original entries if we lose power during the append.



Note that littlefs doesn't maintain a checksum for each entry. Many logging

filesystems do this, but it limits what you can update in a single atomic operation. What we can do instead is group multiple entries into a commit that shares a single checksum. This lets us update multiple unrelated pieces of metadata as long as they reside on the same metadata pair.

commit B and A'

revision 1	revision 0	=>	revision 1	revision 0
A			A	
checksum			checksum	
			B	
v			A'	
			checksum	

- If our block *is* full of entries, we need to somehow remove outdated entries to make space for new ones. This process is called garbage collection, but because littlefs has multiple garbage collectors, we also call this specific case compaction.

Compared to other filesystems, littlefs's garbage collector is relatively simple. We want to avoid RAM consumption, so we use a sort of brute force solution where for each entry we check to see if a newer entry has been written. If the entry is the most recent we append it to our new block. This is where having two blocks becomes important, if we lose power we still have everything in our original block.

During this compaction step we also erase the metadata block and increment the revision count. Because we can commit multiple entries at once, we can write all of these changes to the second block without worrying about power loss. It's only when the commit's checksum is written that the compacted entries and revision count become committed and readable.

commit B', need to compact

revision 1	revision 0	=>	revision 1	revision 2
A			A	A'
checksum			checksum	B'
B			B	checksum
A'			A'	
checksum			checksum	
			v	
			checksum	

- If our block *isn't* full of entries *and* we can't find any garbage, then what?

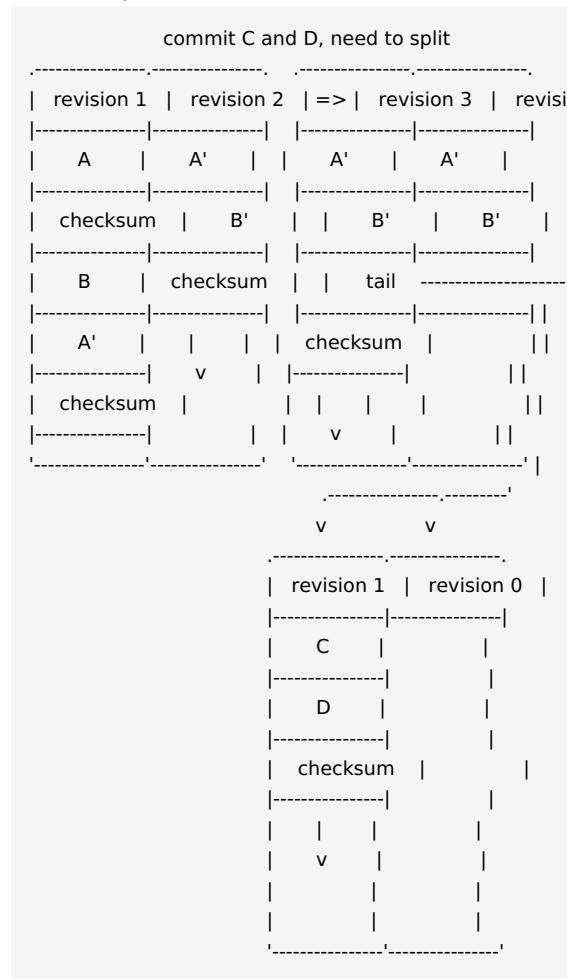
At this point, most logging filesystems would return an error indicating no more space is available, but because we have small logs, overflowing a log isn't really an error condition.

Instead, we split our original metadata pair into two metadata pairs, each containing half of the entries, connected by a tail pointer. Instead of increasing the size of the log and dealing with the scalability issues associated with larger logs, we form a linked list of small bounded logs.

This is a tradeoff as this approach does use more storage space, but at the

benefit of improved scalability.

Despite writing to two metadata pairs, we can still maintain power resilience during this split step by first preparing the new metadata pair, and then inserting the tail pointer during the commit to the original metadata pair.



There is another complexity that crops up when dealing with small logs. The amortized runtime cost of garbage collection is not only dependent on its one time cost ($O(n^2)$ for littlefs), but also depends on how often garbage collection occurs.

Consider two extremes:

1. Log is empty, garbage collection occurs once every n updates
 2. Log is full, garbage collection occurs **every** update

Clearly we need to be more aggressive than waiting for our metadata pair to be full. As the metadata pair approaches fullness the frequency of compactions grows very rapidly.

Looking at the problem generically, consider a log with bytes for each entry, dynamic entries (entries that are outdated during garbage collection), and static entries (entries that need to be copied during garbage collection). If we look at the amortized runtime complexity of updating this log we get this formula:

If we let

be the ratio of static space to the size of our log in bytes, we

find an alternative representation of the number of static and dynamic entries:

Substituting these in for

and

gives us a nice formula for the cost of

updating an entry given how full the log is:

Assuming 100 byte entries in a 4 KiB log, we can graph this using the entry size to find a multiplicative cost:

So at 50% usage, we're seeing an average of 2x cost per update, and at 75% usage, we're already at an average of 4x cost per update.

To avoid this exponential growth, instead of waiting for our metadata pair to be full, we split the metadata pair once we exceed 50% capacity. We do this lazily, waiting until we need to compact before checking if we fit in our 50% limit. This limits the overhead of garbage collection to 2x the runtime cost, giving us an amortized runtime complexity of $O(1)$.

If we look at metadata pairs and linked-lists of metadata pairs at a high level, they have fairly nice runtime costs. Assuming n metadata pairs, each containing m metadata entries, the *lookup* cost for a specific entry has a worst case runtime complexity of $O(nm)$. For *updating* a specific entry, the worst case complexity is $O(nm^2)$, with an amortized complexity of only $O(nm)$.

However, splitting at 50% capacity does mean that in the best case our metadata pairs will only be 1/2 full. If we include the overhead of the second block in our metadata pair, each metadata entry has an effective storage cost of 4x the original size. I imagine users would not be happy if they found that they can only use a quarter of their original storage. Metadata pairs provide a mechanism for performing atomic updates, but we need a separate mechanism for storing the bulk of our data.

CTZ skip-lists

Metadata pairs provide efficient atomic updates but unfortunately have a large storage cost. But we can work around this storage cost by only using the metadata pairs to store references to more dense, copy-on-write (COW) data structures.

[Copy-on-write data structures](#), also called purely functional data structures, are a category of data structures where the underlying elements are immutable. Making changes to the data requires creating new elements containing a copy of the updated data and replacing any references with references to the new elements. Generally, the performance of a COW data structure depends on how many old elements can be reused after replacing parts of the data.

littlefs has several requirements of its COW structures. They need to be efficient to read and write, but most frustrating, they need to be traversable with a constant amount of RAM. Notably this rules out

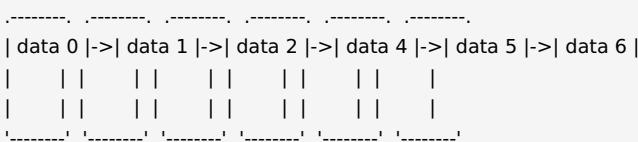
[B-trees](#), which can not be traversed with constant RAM, and

[B+-trees](#), which are not possible to update with COW

operations.

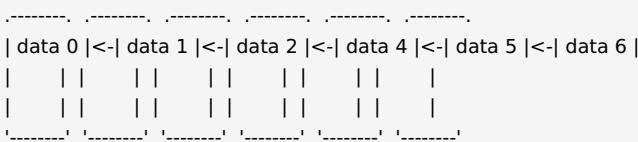
So, what can we do? First let's consider storing files in a simple COW linked-list. Appending a block, which is the basis for writing files, means we have to update the last block to point to our new block. This requires a COW operation, which means we need to update the second-to-last block, and then the third-to-last, and so on until we've copied out the entire file.

A linked-list



To avoid a full copy during appends, we can store the data backwards. Appending blocks just requires adding the new block and no other blocks need to be updated. If we update a block in the middle, we still need to copy the following blocks, but can reuse any blocks before it. Since most file writes are linear, this design gambles that appends are the most common type of data update.

A backwards linked-list



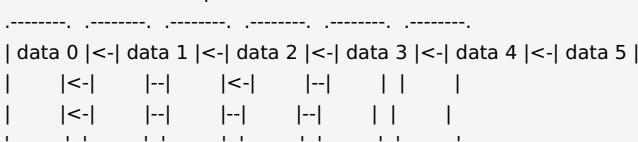
However, a backwards linked-list does have a rather glaring problem. Iterating over a file *in order* has a runtime cost of $O(n^2)$. A quadratic runtime just to read a file! That's awful.

Fortunately we can do better. Instead of a singly linked list, littlefs uses a multilayered linked-list often called a [skip-list](#). However, unlike the most common type of skip-list, littlefs's skip-lists are strictly deterministic built around some interesting properties of the count-trailing-zeros (CTZ) instruction.

The rules CTZ skip-lists follow are that for every n th block where n is divisible by 2^x , that block contains a pointer to block $n-2^x$. This means that each block contains anywhere from 1 to $\log_2 n$ pointers that skip to different preceding elements of the skip-list.

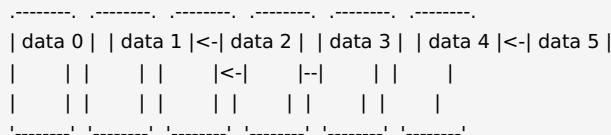
The name comes from heavy use of the [CTZ instruction](#), which lets us calculate the power-of-two factors efficiently. For a give block n , that block contains $\text{ctz}(n)+1$ pointers.

A backwards CTZ skip-list

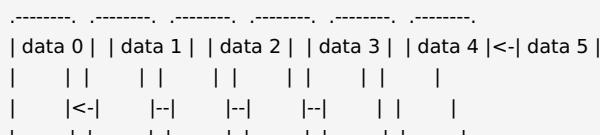


The additional pointers let us navigate the data-structure on disk much more efficiently than in a singly linked list.

Consider a path from data block 5 to data block 1. You can see how data block 3 was completely skipped:



The path to data block 0 is even faster, requiring only two jumps:



We can find the runtime complexity by looking at the path to any block from the block containing the most pointers. Every step along the path divides the search space for the block in half, giving us a runtime of $O(\log n)$.

To get to the block with the most pointers, we can perform the same steps backwards, which puts the runtime at $O(2 \log n) = O(\log n)$. An interesting note is that this optimal path occurs naturally if we greedily choose the pointer that covers the most distance without passing our target.

So now we have a [COW](#) data structure that is cheap to append with a runtime of $O(1)$, and can be read with a worst case runtime of $O(n \log n)$. Given that this runtime is also divided by the amount of data we can store in a block, this cost is fairly reasonable.

This is a new data structure, so we still have several questions. What is the storage overhead? Can the number of pointers exceed the size of a block? How do we store a CTZ skip-list in our metadata pairs?

To find the storage overhead, we can look at the data structure as multiple linked-lists. Each linked-list skips twice as many blocks as the previous, or from another perspective, each linked-list uses half as much storage as the previous. As we approach infinity, the storage overhead forms a geometric series. Solving this tells us that on average our storage overhead is only 2 pointers per block.

Because our file size is limited the word width we use to store sizes, we can also solve for the maximum number of pointers we would ever need to store in a block. If we set the overhead of pointers equal to the block size, we get the following equation. Note that both a smaller block size () and larger word width () result in more storage overhead.

Solving the equation for gives us the minimum block size for some common word widths:

1. 32-bit CTZ skip-list => minimum block size of 104 bytes
2. 64-bit CTZ skip-list => minimum block size of 448 bytes

littlefs uses a 32-bit word width, so our blocks can only overflow with pointers if they are smaller than 104 bytes. This is an easy requirement, as in practice, most block sizes start at 512 bytes. As long as our block size is larger than 104 bytes, we can avoid the extra logic needed to handle pointer overflow.

This last question is how do we store CTZ skip-lists? We need a pointer to the head block, the size of the skip-list, the index of the head block, and our offset in the head block. But it's worth noting that each size maps to a unique index + offset pair. So in theory we can store only a single pointer and size.

However, calculating the index + offset pair from the size is a bit complicated. We can start with a summation that loops through all of the blocks up until our given size. Let
be the block size in bytes,
be the
word width in bits,
be the index of the block in the skip-list, and
be the file size in bytes:

This works quite well, but requires $O(n)$ to compute, which brings the full runtime of reading a file up to $O(n^2 \log n)$. Fortunately, that summation doesn't need to touch the disk, so the practical impact is minimal.

However, despite the integration of a bitwise operation, we can actually reduce this equation to a $O(1)$ form. While browsing the amazing resource that is the [On-Line Encyclopedia of Integer Sequences \(OEIS\)](#), I managed to find [A001511](#), which matches the iteration of the CTZ instruction, and [A005187](#), which matches its partial summation. Much to my surprise, these both result from simple equations, leading us to a rather unintuitive property that ties together two seemingly unrelated bitwise instructions:

where:

1. `ctz()`
) = the number of trailing bits that are 0 in
2. `popcount()`
) = the number of bits that are 1 in

Initial tests of this surprising property seem to hold. As approaches infinity, we end up with an average overhead of 2 pointers, which matches our assumption from earlier. During iteration, the popcount function seems to handle deviations from this average. Of course, just to make sure I wrote a quick script that verified this property for all 32-bit integers.

Now we can substitute into our original equation to find a more efficient equation for file size:

Unfortunately, the popcount function is non-injective, so we can't solve this equation for our index. But what we can do is solve for an index that

is greater than

with error bounded by the range of the `popcount` function.

We can repeatedly substitute

into the original equation until the error

is smaller than our integer resolution. As it turns out, we only need to perform this substitution once, which gives us this formula for our index:

Now that we have our index

, we can just plug it back into the above

equation to find the offset. We run into a bit of a problem with integer

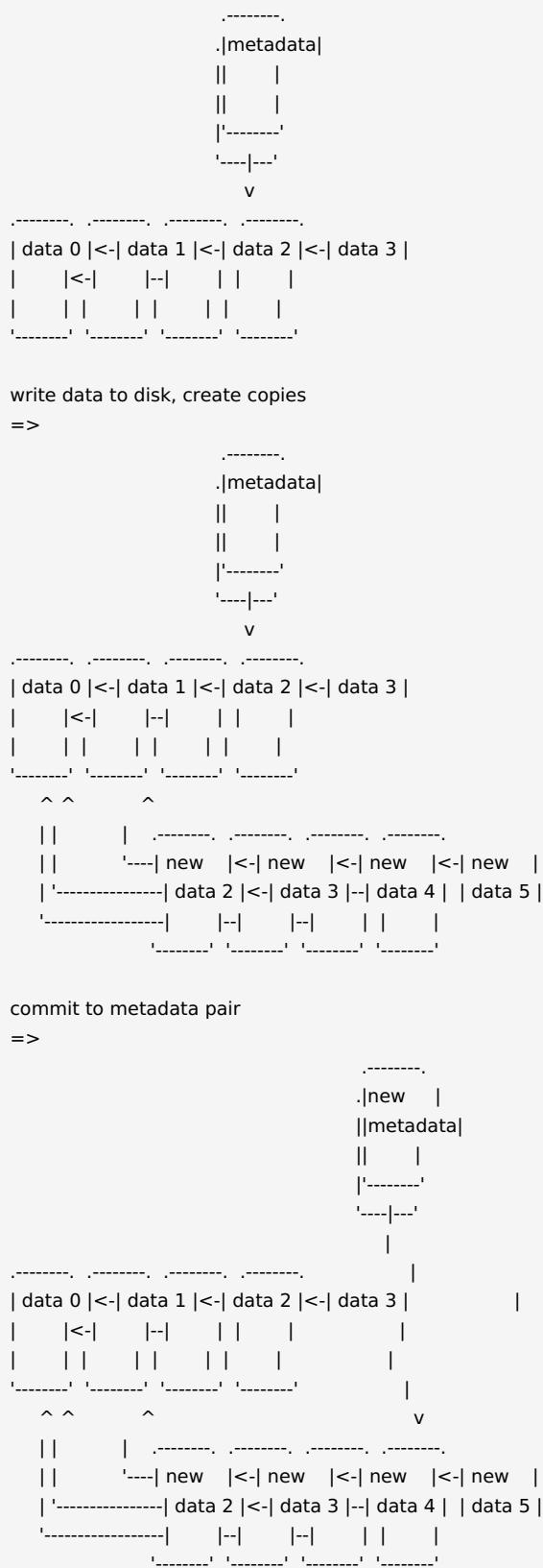
overflow, but we can avoid this by rearranging the equation a bit:

Our solution requires quite a bit of math, but computers are very good at math.

Now we can find both our block index and offset from a size in $O(1)$, letting

us store CTZ skip-lists with only a pointer and size.

CTZ skip-lists give us a COW data structure that is easily traversable in $O(n)$, can be appended in $O(1)$, and can be read in $O(n \log n)$. All of these operations work in a bounded amount of RAM and require only two words of storage overhead per block. In combination with metadata pairs, CTZ skip-lists provide power resilience and compact storage of data.



The block allocator

So we now have the framework for an atomic, wear leveling filesystem. Small two block metadata pairs provide atomic updates, while CTZ skip-lists provide compact storage of data in COW blocks.

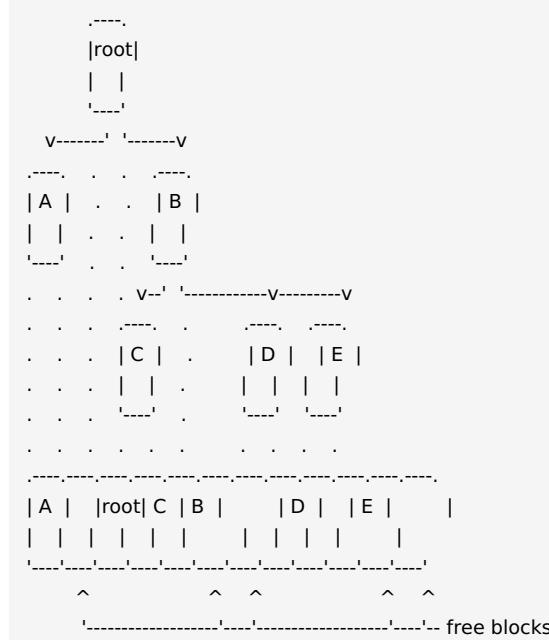
But now we need to look at the [elephant](#) in the room. Where do all these blocks come from?

Deciding which block to use next is the responsibility of the block allocator. In filesystem design, block allocation is often a second-class citizen, but in a COW filesystem its role becomes much more important as it is needed for

nearly every write to the filesystem.

Normally, block allocation involves some sort of free list or bitmap stored on the filesystem that is updated with free blocks. However, with power resilience, keeping these structures consistent becomes difficult. It doesn't help that any mistake in updating these structures can result in lost blocks that are impossible to recover.

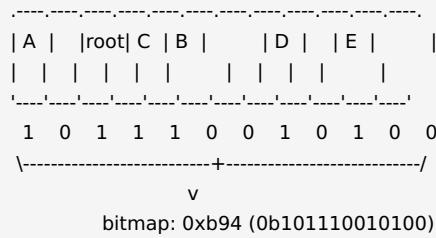
littlefs takes a cautious approach. Instead of trusting a free list on disk, littlefs relies on the fact that the filesystem on disk is a mirror image of the free blocks on the disk. The block allocator operates much like a garbage collector in a scripting language, scanning for unused blocks on demand.



While this approach may sound complicated, the decision to not maintain a free list greatly simplifies the overall design of littlefs. Unlike programming languages, there are only a handful of data structures we need to traverse. And block deallocation, which occurs nearly as often as block allocation, is simply a noop. This "drop it on the floor" strategy greatly reduces the complexity of managing on disk data structures, especially when handling high-risk error conditions.

Our block allocator needs to find free blocks efficiently. You could traverse through every block on storage and check each one against our filesystem tree; however, the runtime would be abhorrent. We need to somehow collect multiple blocks per traversal.

Looking at existing designs, some larger filesystems that use a similar "drop it on the floor" strategy store a bitmap of the entire storage in **RAM**. This works well because bitmaps are surprisingly compact. We can't use the same strategy here, as it violates our constant RAM requirement, but we may be able to modify the idea into a workable solution.



The block allocator in littlefs is a compromise between a disk-sized bitmap and a brute force traversal. Instead of a bitmap the size of storage, we keep track of a small, fixed-size bitmap called the lookahead buffer. During block allocation, we take blocks from the lookahead buffer. If the lookahead buffer is empty, we scan the filesystem for more free blocks, populating our lookahead buffer. In each scan we use an increasing offset, circling the storage as blocks are allocated.

Here's what it might look like to allocate 4 blocks on a decently busy filesystem with a 32 bit lookahead and a total of 128 blocks (512 KiB of storage if blocks are 4 KiB):

```

boot...      lookahead:
           fs blocks: fffff9ffffffffffffefffffff0000
scanning...   lookahead: fffff9ff
           fs blocks: fffff9ffffffffffffefffffff0000
alloc = 21    lookahead: fffffdff
           fs blocks: fffffdffffffffffffefffffff0000
alloc = 22    lookahead: ffffffff
           fs blocks: ffffffffffffefffffffff0000
scanning...   lookahead:     ffffffe
           fs blocks: ffffffffffffefffffffff0000
alloc = 63    lookahead:     fffffff
           fs blocks: ffffffffffffefffffffff0000
scanning...   lookahead:     ffffffff
           fs blocks: ffffffffffffefffffffff0000
scanning...   lookahead:     ffffffrf
           fs blocks: ffffffffffffefffffffff0000
scanning...   lookahead:             ffff0000
           fs blocks: ffffffffffffefffffffff0000
alloc = 112   lookahead:             ffff8000
           fs blocks: ffffffffffffefffffffff0000

```

This lookahead approach has a runtime complexity of $O(n^2)$ to completely scan storage; however, bitmaps are surprisingly compact, and in practice only one or two passes are usually needed to find free blocks. Additionally, the performance of the allocator can be optimized by adjusting the block size or size of the lookahead buffer, trading either write granularity or RAM for allocator performance.

Wear leveling

The block allocator has a secondary role: wear leveling.

Wear leveling is the process of distributing wear across all blocks in the storage to prevent the filesystem from experiencing an early death due to wear on a single block in the storage.

littlefs has two methods of protecting against wear:

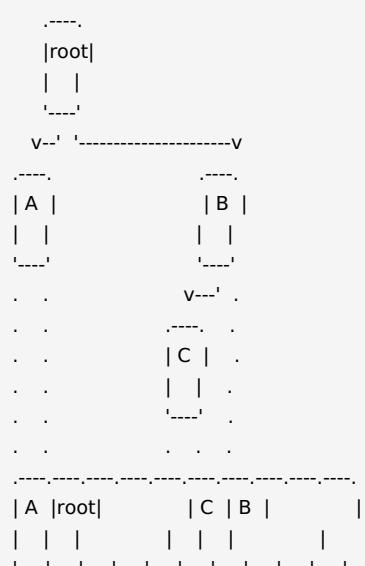
1. Detection and recovery from bad blocks
2. Evenly distributing wear across dynamic blocks

Recovery from bad blocks doesn't actually have anything to do with the block allocator itself. Instead, it relies on the ability of the filesystem to detect and evict bad blocks when they occur.

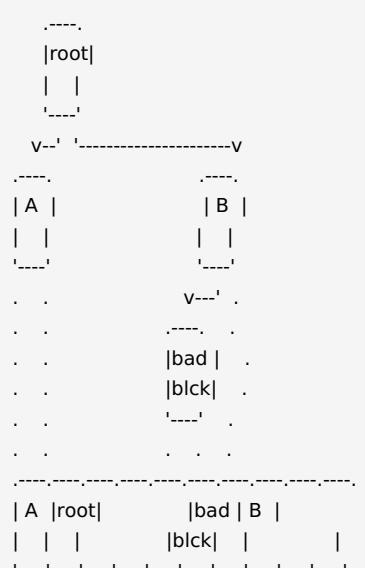
In littlefs, it is fairly straightforward to detect bad blocks at write time. All writes must be sourced by some form of data in RAM, so immediately after we write to a block, we can read the data back and verify that it was written correctly. If we find that the data on disk does not match the copy we have in RAM, a write error has occurred and we most likely have a bad block.

Once we detect a bad block, we need to recover from it. In the case of write errors, we have a copy of the corrupted data in RAM, so all we need to do is evict the bad block, allocate a new, hopefully good block, and repeat the write that previously failed.

The actual act of evicting the bad block and replacing it with a new block is left up to the filesystem's copy-on-bounded-writes (COBW) data structures. One property of COBW data structures is that any block can be replaced during a COW operation. The bounded-writes part is normally triggered by a counter, but nothing prevents us from triggering a COW operation as soon as we find a bad block.

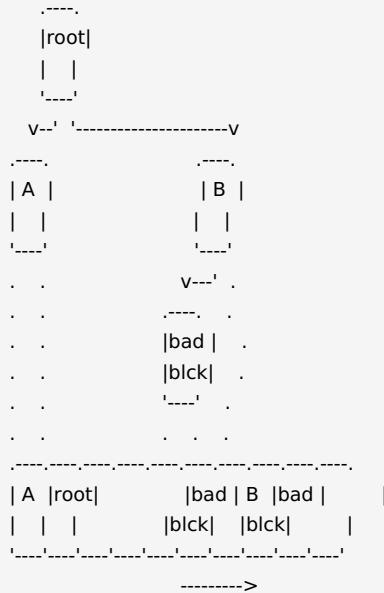


update C
=>



oh no! bad block! relocate C

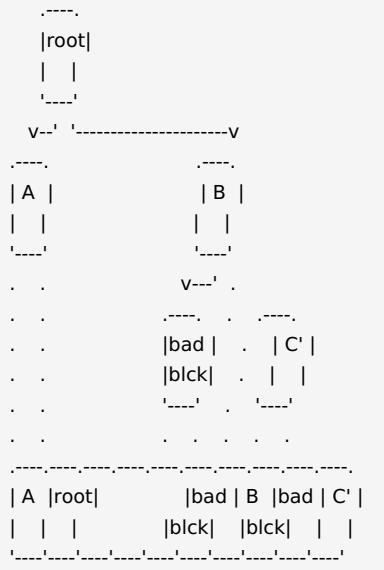
=>



----->

oh no! bad block! relocate C

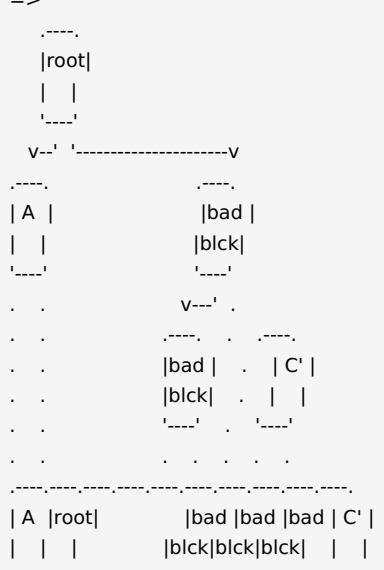
=>



----->

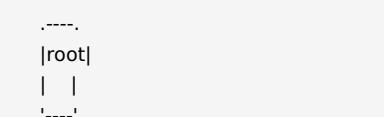
successfully relocated C, update B

=>



oh no! bad block! relocate B

=>



```

v--' -----v
----. . . .
| A |     |bad|     |bad|
|_ _|     |blk|     |blk|
----' . . .
. . . . . . .
. . . |bad| . . |C'|
. . . |blk| . . | |
. . . '---' . . .
. . . . . . .
-----.
| A |root|     |bad|bad|bad| C' |bad|
|_ _ _|     |blk|blk|blk|   |blk|
----->

```

oh no! bad block! relocate B

=>

```

----.
|root|
|_ _|
----'
v--' -----v
----. . . .
| A | | B' |     |bad|
|_ _ | |     |blk|
----' . . .
. . . | . . . .
. . . . . . .
. . . . |bad| . . |C'|
. . . . |blk| . . | |
. . . . '---' . . .
. . . . . . .
-----.
| A |root| B' |     |bad|bad|bad| C' |bad|
|_ _ _|     |blk|blk|blk|   |blk|
----->

```

successfully relocated B, update root

=>

```

----.
|root|
|_ _|
----'
v--' --v
----. . . .
| A | | B' |
|_ _ | | |
----' . . .
. . . . . . .
. . . . | C' |
. . . . | |
. . . . '---' . . .
. . . . . . .
-----.
| A |root| B' |     |bad|bad|bad| C' |bad|
|_ _ _|     |blk|blk|blk|   |blk|
-----'
```

We may find that the new block is also bad, but hopefully after repeating this cycle we'll eventually find a new block where a write succeeds. If we don't, that means that all blocks in our storage are bad, and we've reached the end of our device's usable life. At this point, littlefs will return an "out of space" error. This is technically true, as there are no more good blocks, but as an added benefit it also matches the error condition expected by users of dynamically sized data.

Read errors, on the other hand, are quite a bit more complicated. We don't have a copy of the data lingering around in RAM, so we need a way to reconstruct the original data even after it has been corrupted. One such mechanism for this is [error-correction-codes \(ECC\)](#).

ECC is an extension to the idea of a checksum. Where a checksum such as CRC can detect that an error has occurred in the data, ECC can detect and actually correct some amount of errors. However, there is a limit to how many errors ECC can detect: the [Hamming bound](#). As the number of errors approaches the Hamming bound, we may still be able to detect errors, but can no longer fix the data. If we've reached this point the block is unrecoverable.

littlefs by itself does **not** provide ECC. The block nature and relatively large footprint of ECC does not work well with the dynamically sized data of filesystems, correcting errors without RAM is complicated, and ECC fits better with the geometry of block devices. In fact, several NOR flash chips have extra storage intended for ECC, and many NAND chips can even calculate ECC on the chip itself.

In littlefs, ECC is entirely optional. Read errors can instead be prevented proactively by wear leveling. But it's important to note that ECC can be used at the block device level to modestly extend the life of a device. littlefs respects any errors reported by the block device, allowing a block device to provide additional aggressive error detection.

To avoid read errors, we need to be proactive, as opposed to reactive as we were with write errors.

One way to do this is to detect when the number of errors in a block exceeds some threshold, but is still recoverable. With ECC we can do this at write time, and treat the error as a write error, evicting the block before fatal read errors have a chance to develop.

A different, more generic strategy, is to proactively distribute wear across all blocks in the storage, with the hope that no single block fails before the rest of storage is approaching the end of its usable life. This is called wear leveling.

Generally, wear leveling algorithms fall into one of two categories:

1. [Dynamic wear leveling](#), where we distribute wear over "dynamic" blocks. This can be accomplished by only considering unused blocks.
2. [Static wear leveling](#), where we distribute wear over both "dynamic" and "static" blocks. To make this work, we need to consider all blocks, including blocks that already contain data.

As a tradeoff for code size and complexity, littlefs (currently) only provides dynamic wear leveling. This is a best effort solution. Wear is not distributed perfectly, but it is distributed among the free blocks and greatly extends the life of a device.

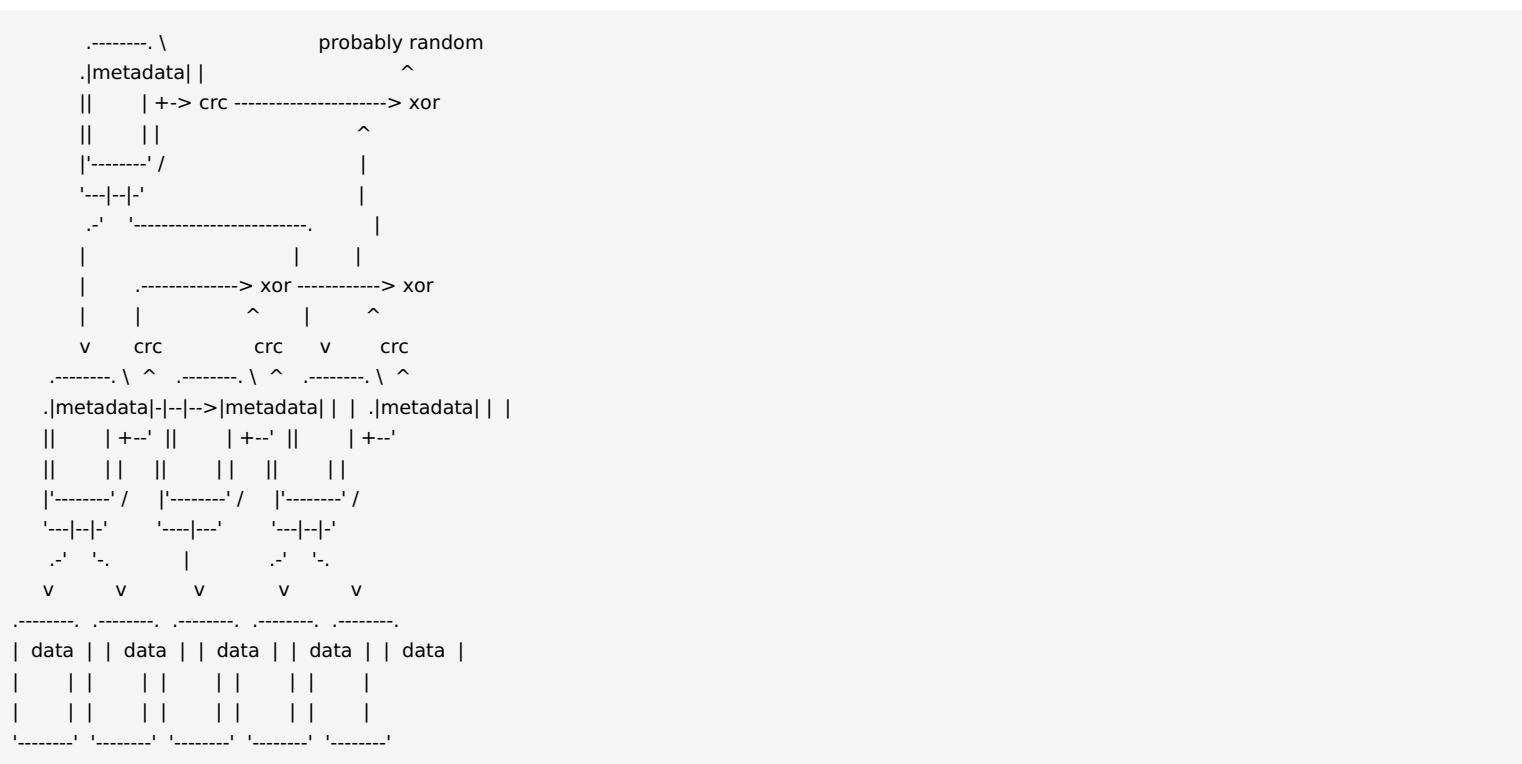
On top of this, littlefs uses a statistical wear leveling algorithm. What this

means is that we don't actively track wear, instead we rely on a uniform distribution of wear across storage to approximate a dynamic wear leveling algorithm. Despite the long name, this is actually a simplification of dynamic wear leveling.

The uniform distribution of wear is left up to the block allocator, which creates a uniform distribution in two parts. The easy part is when the device is powered, in which case we allocate the blocks linearly, circling the device. The harder part is what to do when the device loses power. We can't just restart the allocator at the beginning of storage, as this would bias the wear. Instead, we start the allocator as a random offset every time we mount the filesystem. As long as this random offset is uniform, the combined allocation pattern is also a uniform distribution.

Initially, this approach to wear leveling looks like it creates a difficult dependency on a power-independent random number generator, which must return different random numbers on each boot. However, the filesystem is in a relatively unique situation in that it is sitting on top of a large amount of entropy that persists across power loss.

We can actually use the data on disk to directly drive our random number generator. In practice, this is implemented by xoring the checksums of each metadata pair, which is already calculated to fetch and mount the filesystem.



Note that this random number generator is not perfect. It only returns unique random numbers when the filesystem is modified. This is exactly what we want for distributing wear in the allocator, but means this random number generator is not useful for general use.

Together, bad block detection and dynamic wear leveling provide a best effort solution for avoiding the early death of a filesystem due to wear. Importantly, littlefs's wear leveling algorithm provides a key feature: You can increase the life of a device simply by increasing the size of storage. And if more aggressive wear leveling is desired, you can always combine littlefs with a

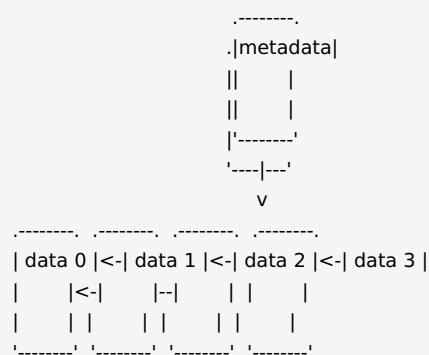
flash translation layer (FTL) to get a small power resilient filesystem with static wear leveling.

Files

Now that we have our building blocks out of the way, we can start looking at our filesystem as a whole.

The first step: How do we actually store our files?

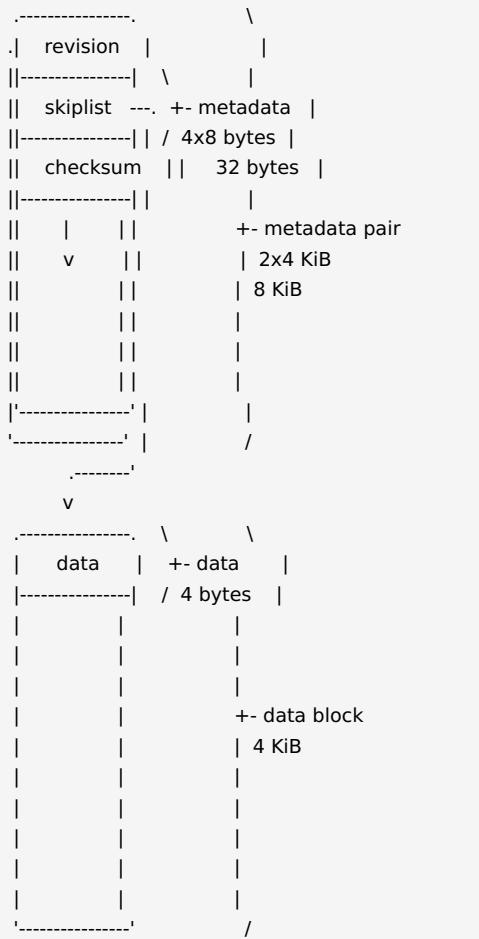
We've determined that CTZ skip-lists are pretty good at storing data compactly, so following the precedent found in other filesystems we could give each file a skip-list stored in a metadata pair that acts as an inode for the file.



However, this doesn't work well when files are small, which is common for embedded systems. Compared to PCs, *all* data in an embedded system is small.

Consider a small 4-byte file. With a two block metadata-pair and one block for the CTZ skip-list, we find ourselves using a full 3 blocks. On most NOR flash with 4 KiB blocks, this is 12 KiB of overhead. A ridiculous 3072x increase.

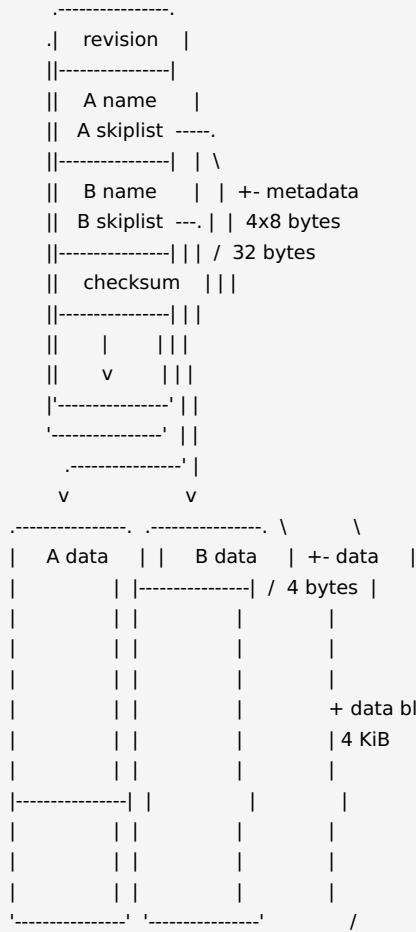
file stored as inode, 4 bytes costs ~12 KiB



We can make several improvements. First, instead of giving each file its own metadata pair, we can store multiple files in a single metadata pair. One way to do this is to directly associate a directory with a metadata pair (or a linked list of metadata pairs). This makes it easy for multiple files to share the directory's metadata pair for logging and reduces the collective storage overhead.

The strict binding of metadata pairs and directories also gives users direct control over storage utilization depending on how they organize their directories.

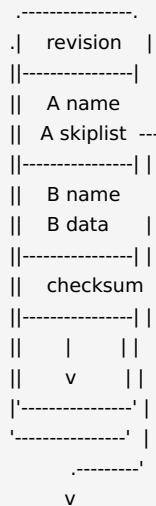
multiple files stored in metadata pair, 4 bytes costs ~4 KiB



The second improvement we can make is noticing that for very small files, our attempts to use CTZ skip-lists for compact storage backfires. Metadata pairs have a ~4x storage cost, so if our file is smaller than 1/4 the block size, there's actually no benefit in storing our file outside of our metadata pair.

In this case, we can store the file directly in our directory's metadata pair. We call this an inline file, and it allows a directory to store many small files quite efficiently. Our previous 4 byte file now only takes up a theoretical 16 bytes on disk.

inline files stored in metadata pair, 4 bytes costs ~16 bytes

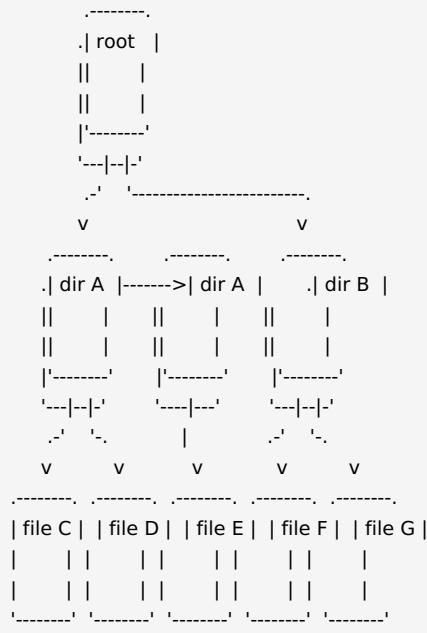


Once the file exceeds 1/4 the block size, we switch to a CTZ skip-list. This means that our files never use more than 4x storage overhead, decreasing as the file grows in size.

Directories

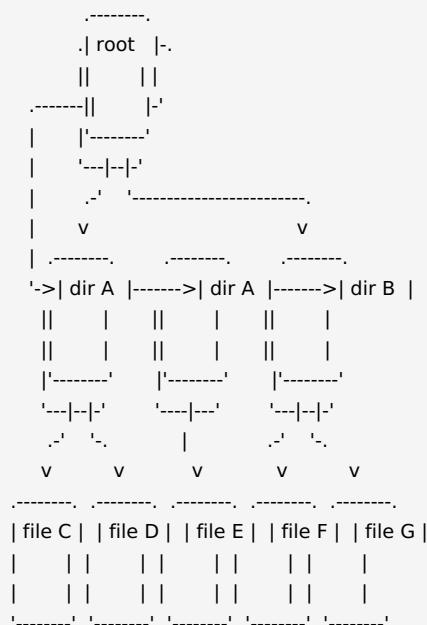
Now we just need directories to store our files. As mentioned above we want a strict binding of directories and metadata pairs, but there are a few complications we need to sort out.

On their own, each directory is a linked-list of metadata pairs. This lets us store an unlimited number of files in each directory, and we don't need to worry about the runtime complexity of unbounded logs. We can store other directory pointers in our metadata pairs, which gives us a directory tree, much like what you find on other filesystems.



The main complication is, once again, traversal with a constant amount of [RAM](#). The directory tree is a tree, and the unfortunate fact is you can't traverse a tree with constant RAM.

Fortunately, the elements of our tree are metadata pairs, so unlike CTZ skip-lists, we're not limited to strict COW operations. One thing we can do is thread a linked-list through our tree, explicitly enabling cheap traversal over the entire filesystem.



Unfortunately, not sticking to pure COW operations creates some problems. Now, whenever we want to manipulate the directory tree, multiple pointers need to be updated. If you're familiar with designing atomic data structures this should set off a bunch of red flags.

To work around this, our threaded linked-list has a bit of leeway. Instead of only containing metadata pairs found in our filesystem, it is allowed to contain metadata pairs that have no parent because of a power loss. These are called orphaned metadata pairs.

With the possibility of orphans, we can build power loss resilient operations that maintain a filesystem tree threaded with a linked-list for traversal.

Adding a directory to our tree:

```
-----.
.| root |-
||   ||
-----||   |-
|   '|-----'
|   '--|---|
|   .-'  '-.
|   v      v
| -----. -----
'->| dir A |->| dir C |
||   ||   ||
||   ||   ||
'|-----'|-----'
'-----' '-----'
```

allocate dir B

=>

```
-----.
.| root |-
||   ||
-----||   |-
|   '|-----'
|   '--|---|
|   .-'  '-.
|   v      v
| -----. -----
'->| dir A |--->| dir C |
||   | .->   |
||   || ||   |
'|-----'|-----'
'-----' |-----'
          |
-----| |
.| dir B |-
||   |
||   |
'|-----'
'-----'
```

insert dir B into threaded linked-list, creating an orphan

=>

```
-----.
.| root |-
||   ||
-----||   |-
|   '|-----'
|   '--|---|
|   .-'  '-----.
|   v      v
| -----. -----
'->| dir A |->| dir B |->| dir C |
||   || || orphan!||   |
||   || ||   ||   |
'|-----'|-----'|-----'
'-----' '-----' '-----'
```

add dir B to parent directory

=>

```
-----.
.| root |-
||   ||
-----||   |-
|   '|-----'
|   '--|---|
|   .-'  '-----.
|   v      v      v
| -----. -----
'->| dir A |->| dir B |->| dir C |
||   || ||   ||   |
||   || ||   ||   |
'|-----'|-----'|-----'
'-----' '-----' '-----'
```

```
||   ||   ||   |
|-----' |-----' |-----'
`-----' `-----'
```

Removing a directory:

```
-----.
.| root |-
||   ||
-----||   |'
|   |'-----'
|   '--|-|-'
|   .----' | '-----.
|   v     v     v
| .-----. .-----. .-----.
'->| dir A |->| dir B |->| dir C |
||   ||   ||   ||
||   ||   ||   ||
|'-----' |'-----' |'-----'
`-----' `-----' `-----'
```

remove dir B from parent directory, creating an orphan

```
=>
-----.
.| root |-
||   ||
-----||   |'
|   |'-----'
|   '--|-|-'
|   .----' .-----.
|   v           v
| .-----. .-----. .-----.
'->| dir A |->| dir B |->| dir C |
||   || orphan!! ||   ||
||   ||   ||   ||
|'-----' |'-----' |'-----'
`-----' `-----' `-----'
```

remove dir B from threaded linked-list, returning dir B to free blocks

```
=>
-----.
.| root |-
||   ||
-----||   |'
|   |'-----'
|   '--|-|-'
|   .----' .
|   v     v
| .-----. .-----.
'->| dir A |->| dir C |
||   ||   ||
||   ||   ||
|'-----' |'-----'
`-----' `-----'
```

In addition to normal directory tree operations, we can use orphans to evict blocks in a metadata pair when the block goes bad or exceeds its allocated erases. If we lose power while evicting a metadata block we may end up with a situation where the filesystem references the replacement block while the threaded linked-list still contains the evicted block. We call this a half-orphan.

```
-----.
.| root |-
||   ||
-----||   |'
|   |'-----'
|   '--|-|-'
```

```
| .----' | '----.  
| v v v  
| .----. .----. .----.  
'->| dir A |->| dir B |->| dir C |  
|| || || || ||  
|| || || || ||  
|'-----' |'-----' |'----'  
'-----' '-----'
```

try to write to dir B

=>

```
.----.  
.| root |-.  
|| ||  
.----.|| |-'  
| |'----'  
| '-|-|-'  
| .----' || '----.  
| v |v v  
| .----. .----. .----.  
'->| dir A |--->| dir B |->| dir C |  
|| |-. | | || ||  
|| || | | || ||  
|'-----' | '----' |'----'  
'-----' | v '----'  
| .----.  
'->| dir B |  
| bad |  
| block! |  
'-----'
```

oh no! bad block detected, allocate replacement

=>

```
.----.  
.| root |-.  
|| ||  
.----.|| |-'  
| |'----'  
| '-|-|-'  
| .----' || '----.  
| v |v v  
| .----. .----. .----.  
'->| dir A |--->| dir B |--->| dir C |  
|| |-. | | |.->|  
|| || | | || ||  
|'-----' | '----' |'----'  
'-----' | v | '----'  
| .----. |  
'->| dir B | |  
| bad | |  
| block! | |  
'-----'|  
|  
| .----. |  
| dir B |--'  
| | |  
| | |  
|-----'
```

insert replacement in threaded linked-list, creating a half-orphan

=>

```
.----.  
.| root |-.  
|| ||  
.----.|| |-'  
| |'----'  
| '-|-|-'  
| .----' || '----.  
| v |v v  
| .----. .----. .----.  
'->| dir A |--->| dir B |--->| dir C |  
|| |-. | | |.->|
```

```

||   || |   || ||
'-----' | '-----' || '-----'
'-----' |   v   | '-----'
| .----- |
| | dir B | |
| | bad   | |
| | block! | |
| '-----' |
| |
| .----- |
'->| dir B |--'
| half  |
| orphan!|
'-----'

```

fix reference in parent directory

=>

```

-----.
.| root |-
||   ||
-----||   |-
|   |'-----'
|   |`--|-|-`-
|   .-----| '-----.
|   v     v     v
| .----- .----- .-----.
'->| dir A |->| dir B |->| dir C |
||   || |   ||   |
||   || |   ||   |
'-----' |'-----' |'-----'
'-----' |'-----' |'-----'

```

Finding orphans and half-orphans is expensive, requiring a $O(n^2)$ comparison of every metadata pair with every directory entry. But the tradeoff is a power resilient filesystem that works with only a bounded amount of RAM. Fortunately, we only need to check for orphans on the first allocation after boot, and a read-only littlefs can ignore the threaded linked-list entirely.

If we only had some sort of global state, then we could also store a flag and avoid searching for orphans unless we knew we were specifically interrupted while manipulating the directory tree (foreshadowing!).

The move problem

We have one last challenge: the move problem. Phrasing the problem is simple:

How do you atomically move a file between two directories?

In littlefs we can atomically commit to directories, but we can't create an atomic commit that spans multiple directories. The filesystem must go through a minimum of two distinct states to complete a move.

To make matters worse, file moves are a common form of synchronization for filesystems. As a filesystem designed for power-loss, it's important we get atomic moves right.

So what can we do?

- We definitely can't just let power-loss result in duplicated or lost files. This could easily break users' code and would only reveal itself in extreme cases. We were only able to be lazy about the threaded linked-list because it isn't user facing and we can handle the corner cases internally.
- Some filesystems propagate COW operations up the tree until a common parent

is found. Unfortunately this interacts poorly with our threaded tree and brings back the issue of upward propagation of wear.

- In a previous version of littlefs we tried to solve this problem by going back and forth between the source and destination, marking and unmarking the file as moving in order to make the move atomic from the user perspective. This worked, but not well. Finding failed moves was expensive and required a unique identifier for each file.

In the end, solving the move problem required creating a new mechanism for sharing knowledge between multiple metadata pairs. In littlefs this led to the introduction of a mechanism called "global state".

Global state is a small set of state that can be updated from *any* metadata pair. Combining global state with metadata pairs' ability to update multiple entries in one commit gives us a powerful tool for crafting complex atomic operations.

How does global state work?

Global state exists as a set of deltas that are distributed across the metadata pairs in the filesystem. The actual global state can be built out of these deltas by xoring together all of the deltas in the filesystem.

```
.|    |->| gdelta |->|      |->| gdelta |->| gdelta |
||    || 0x23  ||  ||      || 0xff  || 0xce  |
||    ||  ||  ||  ||  ||  ||  |
'-----' '-----' '-----' '-----' '-----'
'-----' '-----' '-----' '-----' '-----'
                                         v       v       v
0x00 --> xor -----> xor -----> xor --> gstate 0x12
```

To update the global state from a metadata pair, we take the global state we know and xor it with both our changes and any existing delta in the metadata pair. Committing this new delta to the metadata pair commits the changes to the filesystem's global state.

```

.|  |->| gdelta |->|  |->| gdelta |->| gdelta |
||  || 0x23  ||  ||  || 0xff  || 0xce  |
||  ||  ||  ||  ||  ||  ||  |
'-----' |-----' |-----' |-----' |-----'
'-----' |-----' |-----' |-----' |-----'
           v           v   |       v
0x00 --> xor -----> xor |-----> xor --> gstate = 0x12
                           |
                           |
                           |
change gstate to 0xab --> xor <-----|-----'
=>          |           v
              '-----> xor
                           |
                           v
-----.
.|  |->| gdelta |->|  |->| gdelta |->| gdelta |
||  || 0x23  ||  ||  || 0x46  || 0xce  |
||  ||  ||  ||  ||  ||  ||  |
'-----' |-----' |-----' |-----' |-----'
'-----' |-----' |-----' |-----' |-----'
           v           v   |       v
0x00 --> xor -----> xor -----> xor --> gstate = 0xab

```

To make this efficient, we always keep a copy of the global state in RAM. We only need to iterate over our metadata pairs and build the global state when the filesystem is mounted.

You may have noticed that global state is very expensive. We keep a copy in RAM and a delta in an unbounded number of metadata pairs. Even if we reset the global state to its initial value, we can't easily clean up the deltas on disk. For this reason, it's very important that we keep the size of global state bounded and extremely small. But, even with a strict budget, global state is incredibly valuable.

Now we can solve the move problem. We can create global state describing our move atomically with the creation of the new file, and we can clear this move state atomically with the removal of the old file.

```

----- gstate = no move
.| root |-
||   ||
-----||   |'
|   '|-----'
|   '-|-|-'
|   .----' | '----.
|   v     v     v
| .-----. ,-----. ,----.
'->| dir A |->| dir B |->| dir C |
||   ||   ||   ||   |
||   ||   ||   ||   |
|'-----' |'-----' |'-----'
'---|---' |'-----' |'-----'
      v
.----.
| file D |
|   |
|   |
'-----'

```

begin move, add reference in dir C, change gstate to have move

=>

```

----- gstate = moving file D in dir A (m1)
.| root |-
||   ||
-----||   |'
|   '|-----'
|   '-|-|-'
|   .----' | '----.
|   v     v     v
| .-----. ,-----. ,----.
'->| dir A |->| dir B |->| dir C |
||   ||   ||   || gdelta |
||   ||   ||   ||=m1   |
|'-----' |'-----' |'-----'
'---|---' |'-----' |'---|---'
      |
      v   v
.----.
| file D |
|   |
|   |
'-----'

```

complete move, remove reference in dir A, change gstate to no move

=>

```

----- gstate = no move (m1^~m1)
.| root |-
||   ||
-----||   |'
|   '|-----'
|   '-|-|-'
|   .----' | '----.
|   v     v     v
| .-----. ,-----. ,----.
'->| dir A |->| dir B |->| dir C |
|| gdelta ||   || gdelta |
|| =~m1  ||   || =m1   |
|'-----' |'-----' |'-----'
'---|---' |'-----' |'---|---'
      v
.----.
| file D |
|   |
|   |
'-----'

```

If, after building our global state during mount, we find information
describing an ongoing move, we know we lost power during a move and the file

is duplicated in both the source and destination directories. If this happens, we can resolve the move using the information in the global state to remove one of the files.

```
..... gstate = moving file D in dir A (m1)
.| root |-. ^
|| |-----> xor
|-----|| |-' ^ 
| |-----' | | | | | |
| |-----' |-----' | |
| | | | | | | |
| | .-----> xor -----> xor
| | v | v ^ v ^
| .-----| .-----| .-----|
'->| dir A |->| dir B |->| dir C ||
|| |-' || |-' || gdelta |-
|| | | | | | | =m1 |
|-----' |-----' |-----'
`---|-' |-----' |-----'
 | .-----'
 v v
.
| file D |
| |
| |
`-----'
```

We can also move directories the same way we move files. There is the threaded linked-list to consider, but leaving the threaded linked-list unchanged works fine as the order doesn't really matter.

```

----- gstate = no move (m1^~m1)
.| root |-
||   ||
-----||   |-
|   |'-----'
|   '--|-|-'
|   .----' | '----.
|   v     v     v
| .-----. .-----. .----.
'->| dir A |->| dir B |->| dir C |
|| gdelta ||   || gdelta |
|| =~m1  ||   || =m1  |
|'-----' |'-----' |'-----'
|'-----' |'-----' |'-----'
|           v
| .----.
|   | file D |
|   |   |
|   |   |
|'-----'

```

begin move, add reference in dir C, change gstate to have move

```
=>
----- gstate = moving dir B in root (m1^~m1^m2)
.| root |-
||   ||
-----||   |-
|   |'-----'
|   '--|-|-'
|   .----' | '----.
|   v     |     v
| .-----. |     .----.
'->| dir A |. |     .->| dir C |
|| gdelta ||   || gdelta |
|| =~m1  ||   || =m1^m2 |
|'-----' |   |   ||'-----'
|'-----' |   |   ||'-----'
|   |   .----' |
|   |   v   v   |
|   | .-----. | .----.
|'->| dir B |-' | file D | |
|   ||   ||   |
|   ||   ||   |
|'-----' |'-----'
|'-----'

```

complete move, remove reference in root, change gstate to no move

```
=>
----- gstate = no move (m1^~m1^m2^~m2)
.| root |-
|| gdelta ||
-----|| =~m2  |-
|   |'-----'
|   '--|-|-'
|   .----' '----.
|   v     v
| .-----. .----.
'->| dir A |. .->| dir C |
|| gdelta ||   || gdelta |
|| =~m1  ||   '-|| =m1^m2 |-----.
|'-----' |   |'-----'   | | |
|'-----' |   |'---|-|-'   |
|   |   .-'   '-.   |
|   |   v     v   |
|   | .-----. .----. |
|'->| dir B |--| file D |-
|   ||   ||   |
|   ||   ||   |
|'-----' |'-----'
|'-----'

```

Global state gives us a powerful tool we can use to solve the move problem. And the result is surprisingly performant, only needing the minimum number of states and using the same number of commits as a naive move. Additionally, global state gives us a bit of persistent state we can use for some other small improvements.

Conclusion

And that's littlefs, thanks for reading!

Copyright (c) 2017, Arm Limited. All rights reserved.

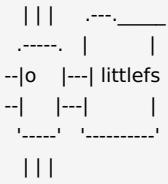
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of ARM nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

littlefs

A little fail-safe filesystem designed for microcontrollers.



Power-loss resilience - littlefs is designed to handle random power failures. All file operations have strong copy-on-write guarantees and if power is lost the filesystem will fall back to the last known good state.

Dynamic wear leveling - littlefs is designed with flash in mind, and provides wear leveling over dynamic blocks. Additionally, littlefs can detect bad blocks and work around them.

Bounded RAM/ROM - littlefs is designed to work with a small amount of memory. RAM usage is strictly bounded, which means RAM consumption does not change as the filesystem grows. The filesystem contains no unbounded

recursion and dynamic memory is limited to configurable buffers that can be provided statically.

Example

Here's a simple example that updates a file named `boot_count` every time main runs. The program can be interrupted at any time without losing track of how many times it has been booted and without corrupting the filesystem:

```
# include "lfs.h"
<p class="mume-header" id="include-lfsh"></p>

// variables used by the filesystem
lfs_t lfs;
lfs_file_t file;

// configuration of the filesystem is provided by this struct
const struct lfs_config cfg = {
    // block device operations
    .read   = user_provided_block_device_read,
    .prog   = user_provided_block_device_prog,
    .erase   = user_provided_block_device_erase,
    .sync   = user_provided_block_device_sync,

    // block device configuration
    .read_size = 16,
    .prog_size = 16,
    .block_size = 4096,
    .block_count = 128,
    .cache_size = 16,
    .lookahead_size = 16,
    .block_cycles = 500,
};

// entry point
int main(void) {
    // mount the filesystem
    int err = lfs_mount(&lfs, &cfg);

    // reformat if we can't mount the filesystem
    // this should only happen on the first boot
    if (err) {
        lfs_format(&lfs, &cfg);
        lfs_mount(&lfs, &cfg);
    }

    // read current count
    uint32_t boot_count = 0;
    lfs_file_open(&lfs, &file, "boot_count", LFS_O_RDWR | LFS_O_CREAT);
    lfs_file_read(&lfs, &file, &boot_count, sizeof(boot_count));

    // update boot count
    boot_count += 1;
    lfs_file_rewind(&lfs, &file);
    lfs_file_write(&lfs, &file, &boot_count, sizeof(boot_count));

    // remember the storage is not updated until the file is closed successfully
    lfs_file_close(&lfs, &file);

    // release any resources we were using
    lfs_unmount(&lfs);

    // print the boot count
    printf("boot_count: %d\n", boot_count);
}
```

Usage

Detailed documentation (or at least as much detail as is currently available) can be found in the comments in [lfs.h](#).

littlefs takes in a configuration structure that defines how the filesystem operates. The configuration struct provides the filesystem with the block device operations and dimensions, tweakable parameters that tradeoff memory usage for performance, and optional static buffers if the user wants to avoid dynamic memory.

The state of the littlefs is stored in the `lfs_t` type which is left up to the user to allocate, allowing multiple filesystems to be in use simultaneously. With the `lfs_t` and configuration struct, a user can format a block device or mount the filesystem.

Once mounted, the littlefs provides a full set of POSIX-like file and directory functions, with the deviation that the allocation of filesystem structures must be provided by the user.

All POSIX operations, such as remove and rename, are atomic, even in event of power-loss. Additionally, file updates are not actually committed to the filesystem until sync or close is called on the file.

Other notes

All littlefs calls have the potential to return a negative error code. The errors can be either one of those found in the `enum lfs_error` in [lfs.h](#), or an error returned by the user's block device operations.

In the configuration struct, the `prog` and `erase` function provided by the user may return a `LFS_ERR_CORRUPT` error if the implementation already can detect corrupt blocks. However, the wear leveling does not depend on the return code of these functions, instead all data is read back and checked for integrity.

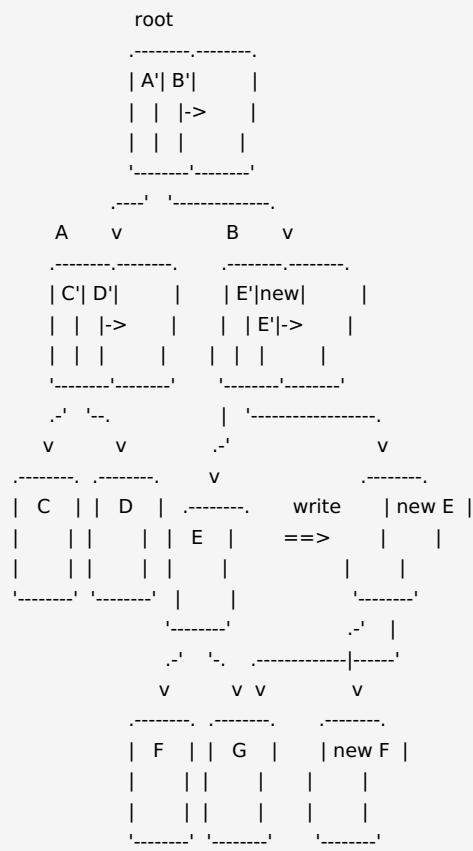
If your storage caches writes, make sure that the provided `sync` function flushes all the data to memory and ensures that the next read fetches the data from memory, otherwise data integrity can not be guaranteed. If the `write` function does not perform caching, and therefore each `read` or `write` call hits the memory, the `sync` function can simply return 0.

Design

At a high level, littlefs is a block based filesystem that uses small logs to store metadata and larger copy-on-write (COW) structures to store file data.

In littlefs, these ingredients form a sort of two-layered cake, with the small logs (called metadata pairs) providing fast updates to metadata anywhere on storage, while the COW structures store file data compactly and without any wear amplification cost.

Both of these data structures are built out of blocks, which are fed by a common block allocator. By limiting the number of erases allowed on a block per allocation, the allocator provides dynamic wear leveling over the entire filesystem.



More details on how littlefs works can be found in [DESIGN.md](#) and [SPEC.md](#).

- [DESIGN.md](#) - A fully detailed dive into how littlefs works.
I would suggest reading it as the tradeoffs at work are quite interesting.
- [SPEC.md](#) - The on-disk specification of littlefs with all the nitty-gritty details. May be useful for tooling development.

Testing

The littlefs comes with a test suite designed to run on a PC using the [emulated block device](#) found in the emubd directory.

The tests assume a Linux environment and can be started with make:

```
make test
```

License

The littlefs is provided under the [BSD-3-Clause](#) license. See [LICENSE.md](#) for more information. Contributions to this project are accepted under the same license.

Individual files contain the following tag instead of the full license text.

```
SPDX-License-Identifier: BSD-3-Clause
```

This enables machine processing of license information based on the SPDX License Identifiers that are here available: <http://spdx.org/licenses/>

Related projects

- [littlefs-fuse](#) - A FUSE wrapper for littlefs. The project allows you to mount littlefs directly on a Linux machine. Can be useful for debugging littlefs if you have an SD card handy.
- [littlefs-js](#) - A javascript wrapper for littlefs. I'm not sure why you would want this, but it is handy for demos. You can see it in action [here](#).
- [mkfs](#) - A command line tool built by the [Lua RTOS](#) guys for making littlefs images from a host PC. Supports Windows, Mac OS, and Linux.
- [Mbed OS](#) - The easiest way to get started with littlefs is to jump into Mbed which already has block device drivers for most forms of embedded storage. littlefs is available in Mbed OS as the [LittleFileSystem](#) class.
- [SPIFFS](#) - Another excellent embedded filesystem for NOR flash. As a more traditional logging filesystem with full static wear-leveling, SPIFFS will likely outperform littlefs on small memories such as the internal flash on microcontrollers.
- [Dhara](#) - An interesting NAND flash translation layer designed for small MCUs. It offers static wear-leveling and power-resilience with only a fixed $O(/address/)$ pointer structure stored on each block and in RAM.

littlefs technical specification

This is the technical specification of the little filesystem. This document covers the technical details of how the littlefs is stored on disk for introspection and tooling. This document assumes you are familiar with the design of the littlefs, for more info on how littlefs works check out [DESIGN.md](#).

```
  |||  .--._____
  .----. |   |
--|o  |---| littlefs |
--|_ |---|   |
'----' '-----'
  |||
```

Some quick notes

- littlefs is a block-based filesystem. The disk is divided into an array of evenly sized blocks that are used as the logical unit of storage.
- Block pointers are stored in 32 bits, with the special value `0xffffffff` representing a null block address.
- In addition to the logical block size (which usually matches the erase block size), littlefs also uses a program block size and read block size. These determine the alignment of block device operations, but don't need to be consistent for portability.
- By default, all values in littlefs are stored in little-endian byte order.

Directories / Metadata pairs

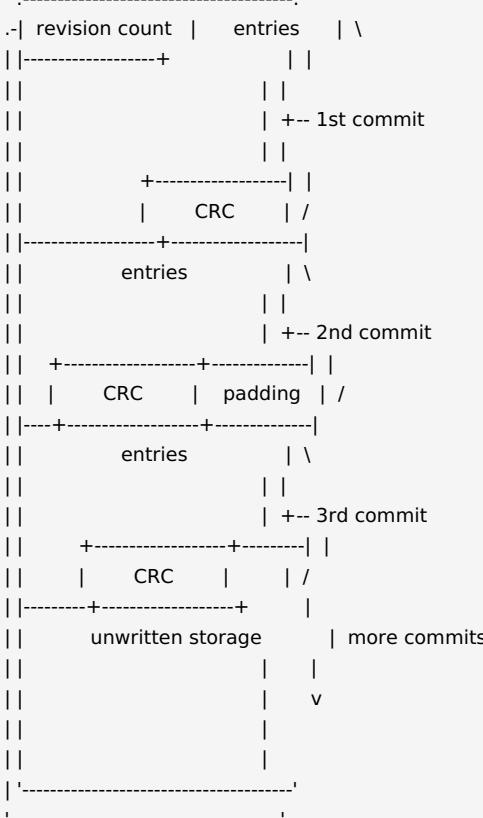
Metadata pairs form the backbone of littlefs and provide a system for distributed atomic updates. Even the superblock is stored in a metadata pair.

As their name suggests, a metadata pair is stored in two blocks, with one block providing a backup during erase cycles in case power is lost. These two blocks are not necessarily sequential and may be anywhere on disk, so a "pointer" to a

metadata pair is stored as two block pointers.

On top of this, each metadata block behaves as an appendable log, containing a variable number of commits. Commits can be appended to the metadata log in order to update the metadata without requiring an erase cycles. Note that successive commits may supersede the metadata in previous commits. Only the most recent metadata should be considered valid.

The high-level layout of a metadata block is fairly simple:



Each metadata block contains a 32-bit revision count followed by a number of commits. Each commit contains a variable number of metadata entries followed by a 32-bit CRC.

Note also that entries aren't necessarily word-aligned. This allows us to store metadata more compactly, however we can only write to addresses that are aligned to our program block size. This means each commit may have padding for alignment.

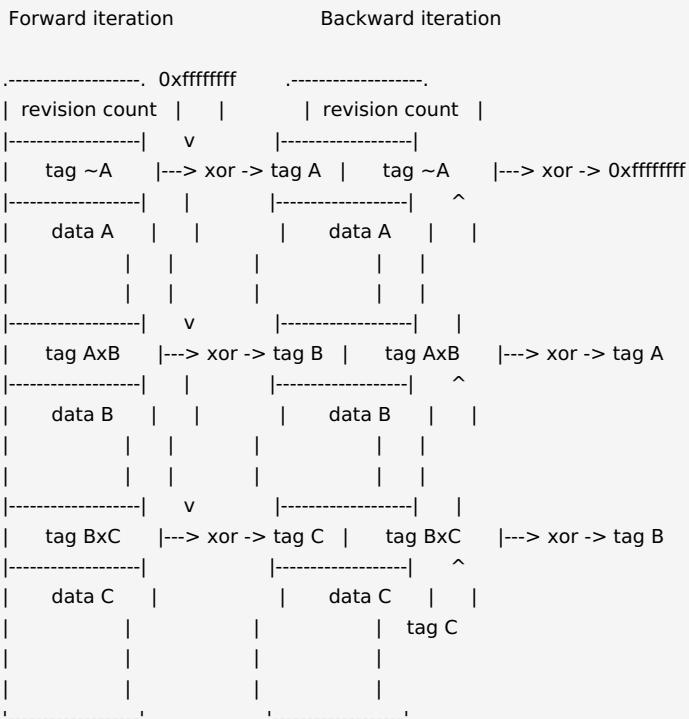
Metadata block fields:

1. **Revision count (32-bits)** - Incremented every erase cycle. If both blocks contain valid commits, only the block with the most recent revision count should be used. Sequence comparison must be used to avoid issues with integer overflow.
2. **CRC (32-bits)** - Detects corruption from power-loss or other write issues. Uses a CRC-32 with a polynomial of `0x04c11db7` initialized with `0xffffffff`.

Entries themselves are stored as a 32-bit tag followed by a variable length blob of data. But exactly how these tags are stored is a little bit tricky.

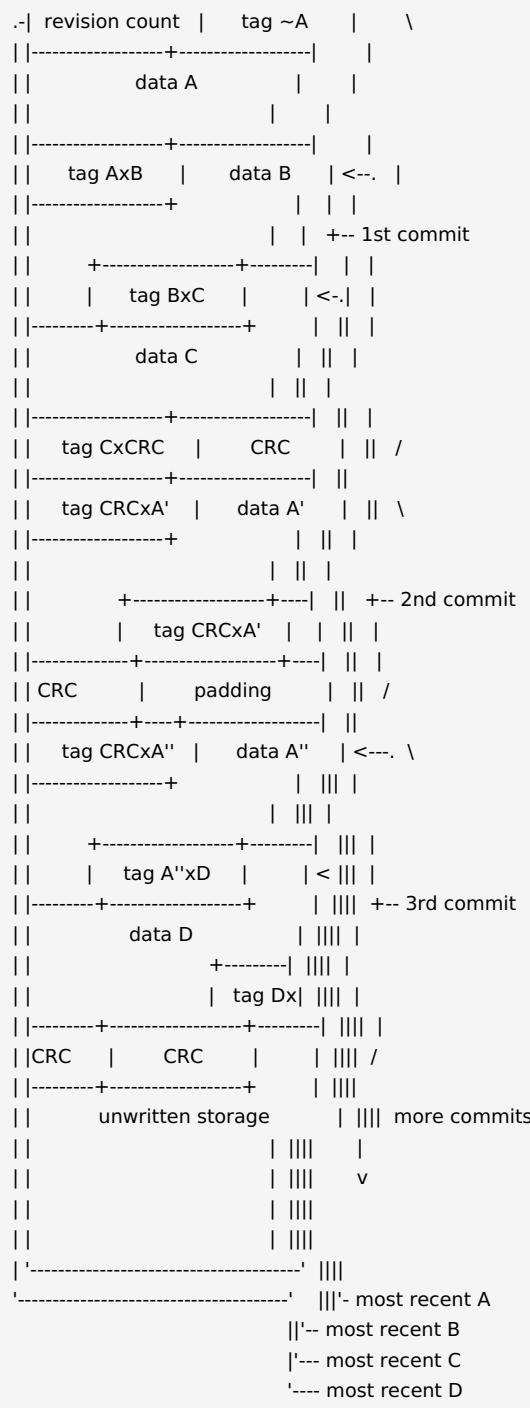
Metadata blocks support both forward and backward iteration. In order to do this without duplicating the space for each tag, neighboring entries have their

tags XORed together, starting with 0xffffffff.



One last thing to note before we get into the details around tag encoding. Each tag contains a valid bit used to indicate if the tag and containing commit is valid. This valid bit is the first bit found in the tag and the commit and can be used to tell if we've attempted to write to the remaining space in the block.

Here's a more complete example of metadata block containing 4 entries:



Metadata tags

So in littlefs, 32-bit tags describe every type of metadata. And this means every type of metadata, including file entries, directory fields, and global state. Even the CRCs used to mark the end of commits get their own tag.

Because of this, the tag format contains some densely packed information. Note that there are multiple levels of types which break down into more info:

```

[----      32      ----]
[1|-- 11  --| 10  --| 10  --]
^   ^   .   ^   ^- length
|.   |   .   '----- id
|.   '----- type (type3)
'----- valid bit
[-3|-- 8 --]
^   ^- chunk
'----- type (type1)

```

Before we go further, there's one important thing to note. These tags are **not** stored in little-endian. Tags stored in commits are actually stored in big-endian (and is the only thing in littlefs stored in big-endian). This little bit of craziness comes from the fact that the valid bit must be the first bit in a commit, and when converted to little-endian, the valid bit finds itself in byte 4. We could restructure the tag to store the valid bit lower, but, because none of the fields are byte-aligned, this would be more complicated than just storing the tag in big-endian.

Another thing to note is that both the tags `0x00000000` and `0xffffffff` are invalid and can be used for null values.

Metadata tag fields:

1. **Valid bit (1-bit)** - Indicates if the tag is valid.
2. **Type3 (11-bits)** - Type of the tag. This field is broken down further into a 3-bit abstract type and an 8-bit chunk field. Note that the value `0x000` is invalid and not assigned a type.
3. **Type1 (3-bits)** - Abstract type of the tag. Groups the tags into 8 categories that facilitate bitmasked lookups.
4. **Chunk (8-bits)** - Chunk field used for various purposes by the different abstract types. `type1+chunk+id` form a unique identifier for each tag in the metadata block.
5. **Id (10-bits)** - File id associated with the tag. Each file in a metadata block gets a unique id which is used to associate tags with that file. The special value `0x3ff` is used for any tags that are not associated with a file, such as directory and global metadata.
6. **Length (10-bits)** - Length of the data in bytes. The special value `0x3ff` indicates that this tag has been deleted.

Metadata types

What follows is an exhaustive list of metadata in littlefs.

0x401 LFS_TYPE_CREATE

Creates a new file with this id. Note that files in a metadata block don't necessarily need a create tag. All a create does is move over any files using this id. In this sense a create is similar to insertion into an imaginary array of files.

The create and delete tags allow littlefs to keep files in a directory ordered alphabetically by filename.

0x4ff LFS_TYPE_DELETE

Deletes the file with this id. An inverse to create, this tag moves over any files neighboring this id similar to a deletion from an imaginary array of files.

0x0xx LFS_TYPE_NAME

Associates the id with a file name and file type.

The data contains the file name stored as an ASCII string (may be expanded to UTF8 in the future).

The chunk field in this tag indicates an 8-bit file type which can be one of the following.

Currently, the name tag must precede any other tags associated with the id and can not be reassigned without deleting the file.

Layout of the name tag:

```
tag          data
[-- 32 --][--- variable length   ---]
[1| 3| 8 | 10 | 10 ][---      (size)      ---]
^ ^ ^ ^ - size      ^- file name
| | | '---- id
| | '----- file type
| '----- type1 (0x0)
'----- valid bit
```

Name fields:

1. **file type (8-bits)** - Type of the file.
2. **file name** - File name stored as an ASCII string.

0x001 LFS_TYPE_REG

Initializes the id + name as a regular file.

How each file is stored depends on its struct tag, which is described below.

0x002 LFS_TYPE_DIR

Initializes the id + name as a directory.

Directories in littlefs are stored on disk as a linked-list of metadata pairs, each pair containing any number of files in alphabetical order. A pointer to the directory is stored in the struct tag, which is described below.

0x0ff LFS_TYPE_SUPERBLOCK

Initializes the id as a superblock entry.

The superblock entry is a special entry used to store format-time configuration and identify the filesystem.

The name is a bit of a misnomer. While the superblock entry serves the same purpose as a superblock found in other filesystems, in littlefs the superblock does not get a dedicated block. Instead, the superblock entry is duplicated across a linked-list of metadata pairs rooted on the blocks 0 and 1. The last metadata pair doubles as the root directory of the filesystem.

```
.| super |-> super |-> super |-> super |-> file B |
|| block | || block | || block | || block | || file C |
||   | ||   | ||   | || file A | || file D |
'-----' '-----' '-----' '-----' '-----'
```

```
\-----+-----/ \-----+-----/
superblock pairs          root directory
```

The filesystem starts with only the root directory. The superblock metadata pairs grow every time the root pair is compacted in order to prolong the life of the device exponentially.

The contents of the superblock entry are stored in a name tag with the superblock type and an inline-struct tag. The name tag contains the magic string "littlefs", while the inline-struct tag contains version and configuration information.

Layout of the superblock name tag and inline-struct tag:

```
tag           data
[-- 32 --][-- 32 --|-- 32 --]
[1|- 11 -| 10 | 10 ][---       64      ---]
^  ^  ^  ^- size (8)    ^- magic string ("littlefs")
|  |  '---- id (0)
|  '----- type (0x0ff)
'----- valid bit
```

```
tag           data
[-- 32 --][-- 32 --|-- 32 --|-- 32 --]
[1|- 11 -| 10 | 10 ][-- 32 --|-- 32 --|-- 32 --]
^  ^  ^  ^  ^- version    ^- block size  ^- block count
|  |  |  |  [-- 32 --|-- 32 --|-- 32 --]
|  |  |  |  [-- 32 --|-- 32 --|-- 32 --]
|  |  |  |  ^- name max   ^- file max   ^- attr max
|  |  |  '- size (24)
|  |  '---- id (0)
|  '----- type (0x201)
'----- valid bit
```

Superblock fields:

1. **Magic string (8-bytes)** - Magic string indicating the presence of littlefs on the device. Must be the string "littlefs".
2. **Version (32-bits)** - The version of littlefs at format time. The version is encoded in a 32-bit value with the upper 16-bits containing the major version, and the lower 16-bits containing the minor version.
This specification describes version 2.0 (0x00020000).
3. **Block size (32-bits)** - Size of the logical block size used by the filesystem in bytes.
4. **Block count (32-bits)** - Number of blocks in the filesystem.
5. **Name max (32-bits)** - Maximum size of file names in bytes.
6. **File max (32-bits)** - Maximum size of files in bytes.
7. **Attr max (32-bits)** - Maximum size of file attributes in bytes.

The superblock must always be the first entry (id 0) in a metadata pair as well as be the first entry written to the block. This means that the superblock entry can be read from a device using offsets alone.

0x2xx LFS_TYPE_STRUCT

Associates the id with an on-disk data structure.

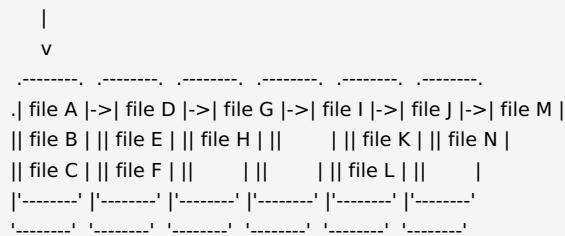
The exact layout of the data depends on the data structure type stored in the chunk field and can be one of the following.

Any type of struct supersedes all other structs associated with the id. For example, appending a ctz-struct replaces an inline-struct on the same file.

0x200 LFS_TYPE_DIRSTRUCT

Gives the id a directory data structure.

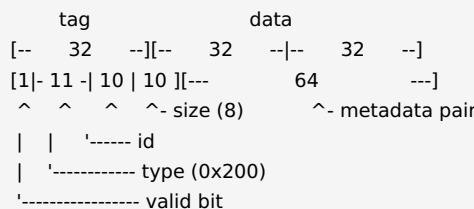
Directories in littlefs are stored on disk as a linked-list of metadata pairs, each pair containing any number of files in alphabetical order.



The dir-struct tag contains only the pointer to the first metadata-pair in the directory. The directory size is not known without traversing the directory.

The pointer to the next metadata-pair in the directory is stored in a tail tag, which is described below.

Layout of the dir-struct tag:



Dir-struct fields:

1. **Metadata pair (8-bytes)** - Pointer to the first metadata-pair in the directory.

0x201 LFS_TYPE_INLINESTRUCT

Gives the id an inline data structure.

Inline structs store small files that can fit in the metadata pair. In this case, the file data is stored directly in the tag's data area.

Layout of the inline-struct tag:

```

tag          data
[-- 32 --][-- variable length   --]
[1|- 11 -| 10 | 10 ][--      (size)      --]
^  ^  ^  ^- size      ^- inline data
|  | '---- id
| '----- type (0x201)
'----- valid bit

```

Inline-struct fields:

1. **Inline data** - File data stored directly in the metadata-pair.

0x202 LFS_TYPE_C TZSTRUCT

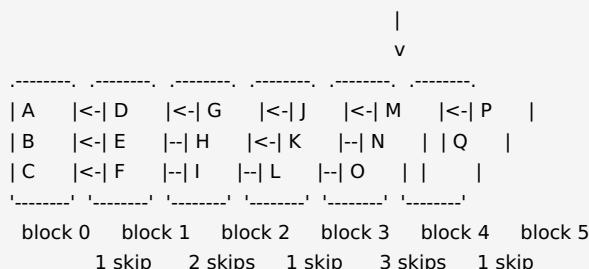
Gives the id a CTZ skip-list data structure.

CTZ skip-lists store files that can not fit in the metadata pair. These files are stored in a skip-list in reverse, with a pointer to the head of the skip-list. Note that the head of the skip-list and the file size is enough information to read the file.

How exactly CTZ skip-lists work is a bit complicated. A full explanation can be found in the [DESIGN.md](#).

A quick summary: For every n th block where n is divisible by 2^x , that block contains a pointer to block $n-2^x$.

These pointers are stored in increasing order of x in each block of the file before the actual data.



Note that the maximum number of pointers in a block is bounded by the maximum file size divided by the block size. With 32 bits for file size, this results in a minimum block size of 104 bytes.

Layout of the CTZ-struct tag:

```

tag          data
[-- 32 --][-- 32 --|-- 32 --]
[1|- 11 -| 10 | 10 ][-- 32 --|-- 32 --]
^  ^  ^  ^      ^- file size
|  |  |  |      '----- file head
|  |  | '--- size (8)
|  |  '---- id
| '----- type (0x202)
'----- valid bit

```

CTZ-struct fields:

1. **File head (32-bits)** - Pointer to the block that is the head of the file's CTZ skip-list.
2. **File size (32-bits)** - Size of the file in bytes.

0x3xx LFS_TYPE_USERATTR

Attaches a user attribute to an id.

littlefs has a concept of "user attributes". These are small user-provided attributes that can be used to store things like timestamps, hashes, permissions, etc.

Each user attribute is uniquely identified by an 8-bit type which is stored in the chunk field, and the user attribute itself can be found in the tag's data.

There are currently no standard user attributes and a portable littlefs implementation should work with any user attributes missing.

Layout of the user-attr tag:

```
    tag          data
[-- 32  --][--- variable length   ---]
[1| 3| 8 | 10 | 10 ][---      (size)      ---]
^ ^ ^ ^ ^- size      ^- attr data
| | | '---- id
| | '----- attr type
| '----- type1 (0x3)
'----- valid bit
```

User-attr fields:

1. **Attr type (8-bits)** - Type of the user attributes.
2. **Attr data** - The data associated with the user attribute.

0x6xx LFS_TYPE_TAIL

Provides the tail pointer for the metadata pair itself.

The metadata pair's tail pointer is used in littlefs for a linked-list containing all metadata pairs. The chunk field contains the type of the tail, which indicates if the following metadata pair is a part of the directory (hard-tail) or only used to traverse the filesystem (soft-tail).

```
-----.
.| dir A |-
||softtail|
-----|     |'
|     '|-----'
|     '--|--|'
|     .-'  -----.
|     v       v
|     -----. -----. -----.
'->| dir B |->| dir B |->| dir C |
||hardtail||softtail||  |
||  ||  ||  ||  |
|-----'|-----'|-----'
|-----'|-----'|-----'
```

Currently any type supersedes any other preceding tails in the metadata pair, but this may change if additional metadata pair state is added.

A note about the metadata pair linked-list: Normally, this linked-list contains every metadata pair in the filesystem. However, there are some operations that

can cause this linked-list to become out of sync if a power-loss were to occur.

When this happens, littlefs sets the "sync" flag in the global state. How exactly this flag is stored is described below.

When the sync flag is set:

1. The linked-list may contain an orphaned directory that has been removed in the filesystem.
2. The linked-list may contain a metadata pair with a bad block that has been replaced in the filesystem.

If the sync flag is set, the threaded linked-list must be checked for these errors before it can be used reliably. Note that the threaded linked-list can be ignored if littlefs is mounted read-only.

Layout of the tail tag:

```
tag          data
[-- 32 --][-- 32 --|-- 32 --]
[1| 3| 8 | 10 | 10 ][---       64      ---]
^ ^ ^ ^ ^- size (8)      ^- metadata pair
| | | '---- id
| | '----- tail type
| '----- type1 (0x6)
'----- valid bit
```

Tail fields:

1. **Tail type (8-bits)** - Type of the tail pointer.
2. **Metadata pair (8-bytes)** - Pointer to the next metadata-pair.

0x600 LFS_TYPE_SOFTTAIL

Provides a tail pointer that points to the next metadata pair in the filesystem.

In this case, the next metadata pair is not a part of our current directory and should only be followed when traversing the entire filesystem.

0x601 LFS_TYPE_HARDDTAIL

Provides a tail pointer that points to the next metadata pair in the directory.

In this case, the next metadata pair belongs to the current directory. Note that because directories in littlefs are sorted alphabetically, the next metadata pair should only contain filenames greater than any filename in the current pair.

0x7xx LFS_TYPE_GSTATE

Provides delta bits for global state entries.

littlefs has a concept of "global state". This is a small set of state that can be updated by a commit to *any* metadata pair in the filesystem.

The way this works is that the global state is stored as a set of deltas distributed across the filesystem such that the global state can be found by the xor-sum of these deltas.

```
.....-----.
.| |->| gdelta |->| |->| gdelta |->| gdelta |
|| || 0x23 || || || 0xff || || 0xce ||
|| || || || || || || |
'-----' |-----' |-----' |-----' |-----'
'-----' |-----' |-----' |-----' |-----'
           v         v         v
0x00 --> xor -----> xor -----> xor --> gstate = 0x12
```

Note that storing globals this way is very expensive in terms of storage usage, so any global state should be kept very small.

The size and format of each piece of global state depends on the type, which is stored in the chunk field. Currently, the only global state is move state, which is outlined below.

0x7ff LFS_TYPE_MOVESTATE

Provides delta bits for the global move state.

The move state in littlefs is used to store info about operations that could cause to filesystem to go out of sync if the power is lost. The operations where this could occur is moves of files between metadata pairs and any operation that changes metadata pairs on the threaded linked-list.

In the case of moves, the move state contains a tag + metadata pair describing the source of the ongoing move. If this tag is non-zero, that means that power was lost during a move, and the file exists in two different locations. If this happens, the source of the move should be considered deleted, and the move should be completed (the source should be deleted) before any other write operations to the filesystem.

In the case of operations to the threaded linked-list, a single "sync" bit is used to indicate that a modification is ongoing. If this sync flag is set, the threaded linked-list will need to be checked for errors before it can be used reliably. The exact cases to check for are described above in the tail tag.

Layout of the move state:

```
tag                      data
[-- 32 --][-- 32 --|--- 32 --|--- 32 --]
[1|- 11 -| 10 | 10 ][1|- 11 -| 10 | 10 |--- 64 ---]
^ ^ ^ ^ ^ ^ ^ ^ padding (0) ^- metadata pair
| | | | | | '---- move id
| | | | | | '----- move type
| | | | | | '----- sync bit
| | | | |
| | | | '- size (12)
| | | | '---- id (0x3ff)
| | | | '----- type (0x7ff)
|----- valid bit
```

Move state fields:

1. **Sync bit (1-bit)** - Indicates if the metadata pair threaded linked-list is in-sync. If set, the threaded linked-list should be checked for errors.

2. **Move type (11-bits)** - Type of move being performed. Must be either `0x000`, indicating no move, or `0x4ff` indicating the source file should be deleted.
 3. **Move id (10-bits)** - The file id being moved.
 4. **Metadata pair (8-bytes)** - Pointer to the metadata-pair containing the move.
-

0x5xx LFS_TYPE_CRC

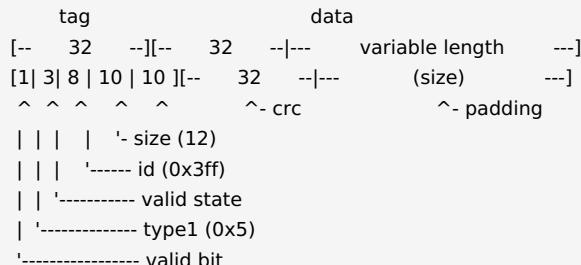
Last but not least, the CRC tag marks the end of a commit and provides a checksum for any commits to the metadata block.

The first 32-bits of the data contain a CRC-32 with a polynomial of `0x04c11db7` initialized with `0xffffffff`. This CRC provides a checksum for all metadata since the previous CRC tag, including the CRC tag itself. For the first commit, this includes the revision count for the metadata block.

However, the size of the data is not limited to 32-bits. The data field may larger to pad the commit to the next program-aligned boundary.

In addition, the CRC tag's chunk field contains a set of flags which can change the behaviour of commits. Currently the only flag in use is the lowest bit, which determines the expected state of the valid bit for any following tags. This is used to guarantee that unwritten storage in a metadata block will be detected as invalid.

Layout of the CRC tag:



CRC fields:

1. **Valid state (1-bit)** - Indicates the expected value of the valid bit for any tags in the next commit.
 2. **CRC (32-bits)** - CRC-32 with a polynomial of `0x04c11db7` initialized with `0xffffffff`.
 3. **Padding** - Padding to the next program-aligned boundary. No guarantees are made about the contents.
-

Mongoose OS support for littlefs

<https://github.com/ARMmbed/littlefs>

Mongoose OS library for MQ135 GAS SENSOR

Getting Started

1. Download and install [Mongoose OS](#).
2. You have a complete [documentation](#) and [video](#) tutorials about Mongoose OS.

Usage of MQ135 Library

Any GPIO pin with ADC capability can be used

```
MQ.MQ135.attach(pin);
```

You can use ambient temperature/humidity correction factor:

```
MQ.MQ135.getCorrectedPPM(temperature, humidity);
```

Note:

- This type of sensor needs 24h after powered to "warning-up" (calibration).

Example of MQ135 Library

See [example](#).

Acknowledgements

Many thanks to the developer of the mgos_stmpe610 library as the this library uses the same structure.

Many thanks to the Mongoose OS developers

ADS7843

Mongoose native SPI driver for ADS7843/XPT2046 Touch Screen Controller

Introduction

ADS7843/XPT2046 is a chip that is connected to a resistive touchscreen.

The touchscreen has resistance in the X and Y directions separately.

When the user touches the touchscreen a resistance in both directions is detected by the touchscreen controller.

The touchscreen controller records the X and Y resistance changes using an ADC that can measure the voltage from the resistance changes in the X and Y directions. A schematic of the test board used is included in this project.

API

This driver initializes the chip and allows the user to register a callback,

which will be called each time the driver senses a touch (TOUCH_DOWN) or a release (TOUCH_UP) event. The callback function will be given the X and Y coordinates where the user pressed the screen. The X and Y coordinates are passed in the mgos_ads7843_event_data structure. The x and y attributes are values in screen pixels.

Details

The SPI bus must be connected to the screen via 5 signals. These are MOSI, MISO, SCK, /CS and /IRQ. When the screen is touched by the user the /IRQ line goes low. In the microcontroller code this causes an interrupt to occur. The interrupt service routine then reads the X and Y ADC values from the ADS7843 device (using the MOSI, MISO, SCK, /CS signals).

The interrupt service routine then converts these ADC values into screen pixel positions. Once this is done a callback function is called. A pointer to a structure that holds the above X and y values along with some other data is passed to the callback function.

Example Application

mos.yml

The driver uses the Mongoose native SPI driver. It is configured by setting up the MOSI, MISO, SCLK pins and assigning one of the three available CS positions, in this example we use CS1:

```
config_schema:
- ["spi.enable", true]
- ["spi.mosi_gpio", 23]
- ["spi.miso_gpio", 19]
- ["spi.sclk_gpio", 18]
- ["spi.cs1_gpio", 27 ] # This defines the ADS7843 SPI chip select pin
- ["ads7843", "o", {title: "ADS7843/XPT2046 TouchScreen"}]
- ["ads7843.cs_index", "i", 1, {title: "spi.cs*_gpio index, 0, 1 or 2"}] # This defines the SPI CS line to use (0, 1 or 2)
- ["ads7843.irq_pin", "i", 25, {title: "IRQ pin (taken low when the display is touched.)"}]
- ["ads7843.x_pixels", "i", 320, {title: "The display pixel count in the horizontal direction"}]
- ["ads7843.y_pixels", "i", 240, {title: "The display pixel count in the vertical direction"}]
- ["ads7843.flip_x", "i", 0, {title: "Flip the X direction (0/1, 0 = no flip). Use this if the x direction is reversed."}]
- ["ads7843.flip_y", "i", 0, {title: "Flip the Y direction (0/1, 0 = no flip). Use this if the y direction is reversed."}]
- ["ads7843.flip_x_y", "i", 0, {title: "Flip the X and Y directions (0/1, 0 = no flip). Use this if the display is upside down."}]
- ["ads7843.min_x_adc", "i", 12, {title: "The min X axis ADC calibration value. Enter the value from debug output (min adc x value at screen edge)."}]
- ["ads7843.max_x_adc", "i", 121, {title: "The max X axis ADC calibration value. Enter the value from debug output (max adc x value at screen edge)."}]
- ["ads7843.min_y_adc", "i", 7, {title: "The min Y axis ADC calibration value. Enter the value from debug output (min adc y value at screen edge)."}]
- ["ads7843.max_y_adc", "i", 118, {title: "The max Y axis ADC calibration value. Enter the value from debug output (max adc y value at screen edge)."}]
```

Application

```

#include "mgos.h"
<p class="mume-header" id="include-mgosh-17"></p>

#include "mgos_ads7843.h"
<p class="mume-header" id="include-mgos_ads7843h"></p>

static void touch_handler(struct mgos_ads7843_event_data *event_data) {
    if (!event_data) {
        return;
    }

    LOG(LL_INFO, ("orientation=%s", event_data->orientation ? "PORTRAIT" : "LANDSCAPE"));
    LOG(LL_INFO, ("Touch %s, down for %.1f seconds", event_data->direction==TOUCH_UP?"UP":"DOWN", event_data->down_seconds));
    LOG(LL_INFO, ("pixels x/y = %d/%d, adc x/y = %d/%d", event_data->x, event_data->y, event_data->x_adc, event_data->y_adc));

}

enum mgos_app_init_result mgos_app_init(void) {

    mgos_ads7843_set_handler(touch_handler);

    return MGOS_APP_INIT_SUCCESS;
}

```

Screen calibration

The min_x_adc, max_x_adc, min_y_adc and max_y_adc should be set in the mos.yml file in order to calibrate the display. This allows the pixel positions returned to accurately represent the position the display was touched.

In order to calibrate the display run the example program and touch each edge of the display.

When running the example application debug data is sent out on the serial port when the display is touched.

E.G

```
[Jan 9 02:36:10.761] touch_handler      orientation=PORTRAIT
[Jan 9 02:36:10.766] touch_handler      Touch DOWN, down for 0.0 seconds
[Jan 9 02:36:10.772] touch_handler      pixels x/y = 0/4, adc x/y = 10/9
```

The above shows the top left corner of the display being touched. The 'adc x/y = 10/9' text contains the min X and Y values. The min_x_adc and min_y_adc attributes in the example mos.yml should be set to the values found.

This should be repeated for the bottom right corner of the display using the max_x_adc and max_y_adc values.

E.G

```
[Jan 9 02:36:15.110] touch_handler      orientation=PORTRAIT
[Jan 9 02:36:15.115] touch_handler      Touch DOWN, down for 0.3 seconds
[Jan 9 02:36:15.120] touch_handler      pixels x/y = 317/233, adc x/y = 120/115
```

Testing

This driver has been tested with the following display

http://www.lcdwiki.com/2.8inch_SPI_Module_ILI9341_SKU:MSP2807

in all four orientations using a modified version of the following project

Google IoT Core integration for Mongoose OS

This library implements integration of Mongoose OS with Google IoT Core.

See tutorial at <https://mongoose-os.com/docs/mongoose-os/cloud/google.md>

Neopixel driver for Mongoose OS

RPC Service - Config

This service provides an ability to manage device configuration remotely.

It is required by the `mos config-get` and `mos config-set` commands.

If this library is not included in the app, those commands won't work.

It is possible to call this service programmatically via serial, HTTP/RESTful,

Websocket, MQTT or other transports

(see [RPC section](#)) or use `mos` tool.

Below is a list of exported RPC methods and arguments:

Config.Get

Get device configuration subtree. Arguments:

```
{  
    // Optional. Path to a config object, e.g. `wifi.sta.ssid`.  
    // If not specified, a full configuration tree is returned.  
    "key": "..."  
}
```

Example usage:

```
mos call Config.Get
{
  "http": {
    "enable": true,
    "listen_addr": "80",
    ...
mos call Config.Get '{"key": "wifi.sta.enable"}'
true
```

This RPC command has a shortcut: `mos config-get`:

```
mos config-get
{
  "http": {
    "enable": true,
    "listen_addr": "80",
    ...
mos config-get wifi.sta.enable
true
```

Config.Set

Set device configuration parameters. Arguments:

```
{
  // Required. Contains a sparse object with configuration parameters.
  // These parameters are applied on top of the existing device configuration.
  "config": { ... }
}
```

Example usage - set `debug.level` to 3:

```
mos call Config.Set '{"config": {"debug": {"level": 3}}}'
```

This RPC command has a shortcut: `mos config-set` which sets the config option, saves it, and reboots the device (since some config options take effect only after reboot):

```
mos config-set debug.level=3
Getting configuration...
Setting new configuration...
Saving and rebooting...
```

Config.Save

Writes an existing device configuration on flash, as a sequence of `confX.json` files (see [description](#)). This makes configuration permanent, preserved after device reboot. Arguments:

```
{
  "reboot": false  // Optional. Whether to reboot the device after the call
}
```

Example usage:

```
mos call Config.Save '{"reboot": true}'
```

Core meta-library

Web UI for WiFi setup

Allows to call local RPC services - implements RPC Loopback channel

This RPC service provides a way to call local RPC services - i.e. those running on the device itself.

Example usage in C/C++:

```
# include "mg_rpc_channel_loopback.h"
<p class="mume-header" id="include-mg_rpc_channel_loopbackh"></p>

struct mg_rpc_call_opts opts = {.dst = mg_mk_str(MGOS_RPC_LOOPBACK_ADDR) };
mg_rpc_callf(mgos_rpc_get_global(), mg_mk_str("My.Func"), NULL, NULL, &opts,
    "{param1: %Q, param2: %d}", "jaja", 1234);
```

Example usage in JavaScript:

```
RPC.call(RPC.LOCAL, 'Config.Save', {reboot: true}, function (resp, ud) {
  print('Response:', JSON.stringify(resp));
}, null);
```

A cross-cloud device shadow API

Overview

This library provides a device shadow interface to manage device metadata on the cloud.

VL53L0X library for Mongoose OS

Version: 1.0

Release date: 2017 Nov 17

Summary

This is a library for the Mongoose OS that helps interface with ST's [VL53L0X time-of-flight distance sensor](#).

The library makes it simple to configure the sensor and read range data from it via I²C.

This library is simple port of pololu's [VL53L0X library for Arduino](#).

Supported platforms

This library is designed to work with Mongoose OS versions 1.19.6 or later;

I have not tested it with earlier versions.

This library should support any boards which support Mongoose OS and I²C.

Getting started

Hardware

I have tested with [ESPr Developer 32](#)

and [VL53L0X Time-of-Flight Range sensor module](#).

Make the following connections between mongoose os board and the VL53L0X module:

ESP32 boards

ESP32	VL53L0X
-----	-----
3V3	- VIN
GND	- GND
SDA(I021)	- SDA
SCL(I022)	- SCL

Software

Add configuration below to your application's mos.yml

```
config_schema:  
- ["i2c.enable", true]  
- ["i2c.sda_gpio", 21]  
- ["i2c.scl_gpio", 22]  
  
libs:  
- origin: https://github.com/mongoose-os-libs/arduino-pololu-vl53l0x
```

Examples

Simple Javascript example is available under examples directory.

ST's VL53L0X API and this library

Most of the functionality of this library is based on the

[VL53L0X API](#)

provided by ST (STSW-IMG005), and some of the explanatory comments in the code are quoted or paraphrased from the API source code, API user manual (UM2039), and the VL53L0X datasheet. For more explanation about the library code and how it was derived from the API, see the comments in VL53L0X.cpp.

This library is intended to provide a quicker and easier way to get started using the VL53L0X with an Arduino-compatible controller, in contrast to customizing and compiling ST's API for the Arduino. The library has a more streamlined interface, as well as smaller storage and memory footprints. However, it does not implement some of the more advanced functionality available in the API (for example, calibrating the sensor to work well under a cover glass), and it has less robust error checking. For advanced applications, especially when storage and memory are less of an issue, consider using the VL53L0X API directly.

Library reference

- `Pololu_VL53L0X.create()`

Constructor.

- `int getLastStatus()`

The status of the last I²C write transmission. See the

[Wire.endTransmission\(\) documentation](#)

for return values.

- `void setAddress(int new_addr)`

Changes the I²C slave device address of the VL53L0X to the given value (7-bit).

- `int getAddress()`

Returns the current I²C address.

- `void begin()`
- `void end()`

Begin or end underlying Arduino's I²C library. Call begin before call init.

- `bool init_2v8()`
- `bool init_1v8()`

Initializes and configures the sensor. If the optional argument `io_2v8` is true (the default if not specified), the sensor is configured for 2V8 mode (2.8 V I/O); if false, the sensor is left in 1V8 mode. The return value is a boolean indicating whether the initialization completed successfully.

- `void writeReg(int reg, int value)`

Writes an 8-bit sensor register with the given value.

Register address constants are defined by the `regAddr` enumeration type in `VL53L0X.h`.

Example use: `sensor.writeReg(VL53L0X::SYSRANGE_START, 0x01);`

- `void writeReg16Bit(int reg, int value)`

Writes a 16-bit sensor register with the given value.

- `void writeReg32Bit(int reg, int value)`

Writes a 32-bit sensor register with the given value.

- `int readReg(int reg)`

Reads an 8-bit sensor register and returns the value read.

- `int readReg16Bit(int reg)`

Reads a 16-bit sensor register and returns the value read.

- `int readReg32Bit(int reg)`

Reads a 32-bit sensor register and returns the value read.

- `void writeMulti(int reg, char const * src, int count)`

Writes an arbitrary number of bytes from the given array to the sensor, starting at the given register.

- `void readMulti(int reg, int * dst, int count)`

Reads an arbitrary number of bytes from the sensor, starting at the given register, into the given array.

- `bool setSignalRateLimit(float limit_Mcps)`

Sets the return signal rate limit to the given value in units of MCPS (mega counts per second). This is the minimum amplitude of the signal reflected from the target and received by the sensor necessary for it to report a valid reading. Setting a lower limit increases the potential range of the sensor but also increases the likelihood of getting an inaccurate reading because of reflections from objects other than the intended target. This limit is initialized to 0.25 MCPS by default. The return value is a boolean indicating whether the requested limit was valid.

- `float getSignalRateLimit()`

Returns the current return signal rate limit in MCPS.

- `bool setMeasurementTimingBudget(int budget_us)`

Sets the measurement timing budget to the given value in microseconds. This is the time allowed for one range measurement; a longer timing budget allows for more accurate measurements. The default budget is about 33000 microseconds, or 33 ms; the minimum is 20 ms. The return value is a boolean indicating whether the requested budget was valid.

- `int getMeasurementTimingBudget()`

Returns the current measurement timing budget in microseconds.

- `bool setVcseIPulsePeriod(vcseIPeriodType type, int period_pclks)`

Sets the VCSEL (vertical cavity surface emitting laser) pulse period for the given period type (`VL53L0X::VcseIPeriodPreRange` or `VL53L0X::VcseIPeriodFinalRange`) to the given value (in PCLKs). Longer periods increase the potential range of the sensor. Valid values are (even numbers only):

Pre: 12 to 18 (initialized to 14 by default)

Final: 8 to 14 (initialized to 10 by default)

The return value is a boolean indicating whether the requested period was valid.

- `int getVcseIPulsePeriod(vcseIPeriodType type)`

Returns the current VCSEL pulse period for the given period type.

- `void startContinuous(int period_ms = 0)`

Starts continuous ranging measurements. If the optional argument `period_ms` is 0 (the default if not specified), continuous back-to-back mode is used (the sensor takes measurements as often as possible); if it is nonzero, continuous timed mode is used, with the specified inter-measurement period in milliseconds determining how often the sensor takes a measurement.

- `void stopContinuous()`

Stops continuous mode.

- `int readRangeContinuousMillimeters()`

Returns a range reading in millimeters when continuous mode is active.

- `int readRangeSingleMillimeters()`

Performs a single-shot ranging measurement and returns the reading in millimeters.

- `void setTimeout(int timeout)`

Sets a timeout period in milliseconds after which read operations will abort if the sensor is not ready. A value of 0 disables the timeout.

- `int getTimeout()`

Returns the current timeout period setting.

- `bool timeoutOccurred()`

Indicates whether a read timeout has occurred since the last call to `timeoutOccurred()`.

Version history

- 1.0 (2017 Nov 17): Original release.
- 1.1 (2017 Nov 29): Avoid "unused-parameter" error.

How to contribute

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

HTU21D(F) I2C Driver

Sensor details

The HTU21D(F) is a new digital humidity sensor with temperature output by MEAS. Setting new standards in terms of size and intelligence, it is embedded in a reflow solderable Dual Flat No leads (DFN) package with a small 3 x 3 x 0.9 mm footprint. This sensor provides calibrated, linearized signals in digital, I2C format.

HTU21D(F) digital humidity sensors are dedicated humidity and temperature plug and play transducers for OEM applications where reliable and accurate measurements are needed. Direct interface with a micro-controller is made possible with the module for humidity and temperature digital outputs. These low power sensors are designed for high volume and cost sensitive applications with tight space constraints.

Every sensor is individually calibrated and tested. Lot identification is printed on the sensor and an electronic identification code is stored on the chip – which can be read out by command. Low battery can be detected and a checksum improves communication reliability. The resolution of these digital humidity sensors can be changed by command (8/12bit up to 12/14bit for RH/T).

With MEAS' improvements and miniaturization of this sensor, the performance-to-price ratio has been improved – and eventually, any device should benefit from its cutting edge energy saving operation mode. Optional PTFE filter/membrane (F) protects HTU21D digital humidity sensors against dust and water immersion, as well as against contamination by particles. PTFE filter/membranes preserve a high response time. The white PTFE filter/membrane is directly stuck on the sensor housing.

See [datasheet](#) for implementation details.

A great place to buy a ready made and tested unit is at [Adafruit](#).

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```

#include "mgos.h"
<p class="mume-header" id="include-mgosh-18"></p>

#include "mgos_i2c.h"
<p class="mume-header" id="include-mgos_i2ch-1"></p>

#include "mgos_htu21df.h"
<p class="mume-header" id="include-mgos_htu21dfh"></p>

static struct mgos_htu21df *s_htu21df;

static void timer_cb(void *user_data) {
    float temperature, humidity;

    temperature=mgos_htu21df_getTemperature(s_htu21df);
    humidity=mgos_htu21df_getHumidity(s_htu21df);

    LOG(LL_INFO, ("htu21df temperature=%.2f humidity=%.2f", temperature, humidity));

    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_i2c *i2c;

    i2c=mgos_i2c_get_global();
    if (!i2c) {
        LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
    } else {
        s_htu21df=mgos_htu21df_create(i2c, 0x40); // Default I2C address
        if (s_htu21df) {
            mgos_set_timer(5000, true, timer_cb, NULL);
        } else {
            LOG(LL_ERROR, ("Could not initialize sensor"));
        }
    }
    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual](#)

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

Si7021 I2C Driver

A Mongoose library for Silicon Si7021 integrated circuit.

Sensor details

The Si7021 I2C Humidity and Temperature Sensor is a monolithic CMOS IC integrating humidity and temperature sensor elements, an analog-to-digital converter, signal processing, calibration data, and an I2C Interface. The patented use of industry-standard, low-K polymeric dielectrics for sensing humidity enables the construction of low-power, monolithic CMOS Sensor ICs with low drift and hysteresis, and excellent long term stability.

The humidity and temperature sensors are factory-calibrated and the calibration data is stored in the on-chip non-volatile memory. This ensures that the sensors are fully interchangeable, with no recalibration or software changes required.

The Si7021 is available in a 3x3 mm DFN package and is reflow solderable. It can be used as a hardware- and software-compatible drop-in upgrade for existing RH/temperature sensors in 3x3 mm DFN-6 packages, featuring precision sensing over a wider range and lower power consumption. The optional factory-installed

cover offers a low profile, convenient means of protecting the sensor during assembly (e.g., reflow soldering) and throughout the life of the product, excluding liquids (hydrophobic/oleophobic) and particulates.

The Si7021 offers an accurate, low-power, factory-calibrated digital solution ideal for measuring humidity, dew-point, and temperature, in applications ranging from HVAC/R and asset tracking to industrial and consumer platforms.

See [datasheet](#)

for implementation details.

A great place to buy a ready made and tested unit is at [Adafruit](#).

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```
# include "mgos.h"
<p class="mume-header " id="include-mgosh-19"></p>

# include "mgos_i2c.h"
<p class="mume-header " id="include-mgos_i2ch-2"></p>

# include "mgos_si7021.h"
<p class="mume-header " id="include-mgos_si7021h"></p>

static struct mgos_si7021 *s_si7021;

static void timer_cb(void *user_data) {
    float temperature, humidity;

    temperature=mgos_si7021_getTemperature(s_si7021);
    humidity=mgos_si7021_getHumidity(s_si7021);

    LOG(LL_INFO, ("si7021 temperature=%.2f humidity=%.2f", temperature, humidity));

    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_i2c *i2c;

    i2c=mgos_i2c_get_global();
    if (!i2c) {
        LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
    } else {
        s_si7021=mgos_si7021_create(i2c, 0x40); // Default I2C address
        if (s_si7021) {
            mgos_set_timer(5000, true, timer_cb, NULL);
        } else {
            LOG(LL_ERROR, ("Could not initialize sensor"));
        }
    }
    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Arduino Adafruit TSL2561 library for Mongoose OS

This is a port of the [Adafruit Arduino driver for the TSL2561 sensor](#) ported to run on the [Mongoose OS ecosystem](#).

Usage is extremely simple....

in **mos.yml**, add to **libs:** section,

```
- origin: https://github.com/mongoose-os-libs/arduino-adafruit-tsl2561
```

in your **init.js**, add something like the following,

```
load('api_arduino_tsl2561.js');
```

and

```
//Initialize Adafruit_TSL2561 library
let tsl = Adafruit_TSL2561.create();
print('Adafruit_TSL2561.TSL2561_GAIN_16X -> ',Adafruit_TSL2561.TSL2561_GAIN_16X);
tsl.setGain(Adafruit_TSL2561.TSL2561_GAIN_16X);
tsl.setIntegrationTime(Adafruit_TSL2561.TSL2561_INTEGRATIONTIME_402MS);
tsl.begin();

let tslGetData = function() {
  let vis = tsl.getVisible();
  let ir = tsl.getInfrared();
  let lux = tsl.calculateLux(vis, ir);
  print('TSL2561: Vis: ', vis, ', IR: ', ir, ', Lux: ', lux);
};

let tslTimer = Timer.set(10000 /* milliseconds */, true /* repeat */, tslGetData, null);
```

to use the library.

Enjoy!

This library provides PWM ([pulse-width modulation](#)) support for Mongoose OS.

A Library of Utility Functions for Mongoose OS apps with Sensors

Overview

A collection of C-functions with JS wrappers that make it convenient to calculate derived quantities (e.g. dew point temperature), convert between different units of measure, and perform simple statistics on arrays of (double precision) values.

Typical usage pattern:

in **mos.yml**, add to **libs:** section,

```
- origin: https://github.com/mongoose-os-libs/sensor-utils
```

in your **init.js**, add something like the following,

```
load('api_sensor_utils.js');
```

and (here is example usage that tests the API),

```

print('Test api_sensor_utils.js          ');

//Initialize a statistics object (handle)

let StatsObj = SensorUtils.initStatistics(4);

// add some numbers (e.g. obtained from sensor readings)

SensorUtils.addDataValue( StatsObj, 5.51 );
SensorUtils.addDataValue( StatsObj, 8.43 );
SensorUtils.addDataValue( StatsObj, 6.77 );
SensorUtils.addDataValue( StatsObj, 4.33 );
SensorUtils.addDataValue( StatsObj, 6.20 );
SensorUtils.addDataValue( StatsObj, 7.18 );
SensorUtils.addDataValue( StatsObj, 4.92 );
SensorUtils.addDataValue( StatsObj, 5.89 );

print('Statistics: ',SensorUtils.calculateStatistics(StatsObj));

// other convenience functions.

print('SensorUtils.fahrenheit(0.0) -> ',SensorUtils.fahrenheit(0.0));
print('SensorUtils.celsius(32.0) -> ',SensorUtils.celsius(32.0));
print('SensorUtils.inchesHg(101325) -> ',SensorUtils.inchesHg(101325));
print('SensorUtils.mmHg(101325) -> ',SensorUtils.mmHg(101325));
print('SensorUtils.atmospheresHg(29.9213) -> ',SensorUtils.atmospheresHg(29.9213));
print('SensorUtils.atmospheresP(101325) -> ',SensorUtils.atmospheresP(101325));
print('SensorUtils.lengthF(1/12) -> ',SensorUtils.lengthF(1.0/12.0));
print('SensorUtils.computeDewpoint(4.6, 50.89) -> ',SensorUtils.computeDewpoint(4.6, 50.89));
print('SensorUtils.computeAltitude(103245.5, 101325.0) -> ',SensorUtils.computeAltitude(103245.5, 101325.0));

```

Executing the above JS code on an ESP32 typically gives the following output:

```

[Jan 17 08:35:35.996] Test api_sensor_utils.js

[Jan 17 08:35:36.183] Statistics: {"samples":8, "length":8, "mean":6.1538, "std_deviation":1.3065, "data":[ 5.5100, 8.4300, 6.7700, 4.3300, 6.2000, 7.1800, 4.9200, 5.8900]}

[Jan 17 08:35:36.214] SensorUtils.fahrenheit(0.0) -> 32
[Jan 17 08:35:36.226] SensorUtils.celsius(32.0) -> 0
[Jan 17 08:35:36.239] SensorUtils.inchesHg(101325) -> 29.921252
[Jan 17 08:35:36.255] SensorUtils.mmHg(101325) -> 1.178002
[Jan 17 08:35:36.268] SensorUtils.atmospheresHg(29.9213) -> 1.000043
[Jan 17 08:35:36.282] SensorUtils.atmospheresP(101325) -> 1
[Jan 17 08:35:36.295] SensorUtils.lengthF((2.54/100.0) * 12) -> 1.000000
[Jan 17 08:35:36.313] SensorUtils.computeDewpoint(4.6, 50.89) -> -4.634330
[Jan 17 08:35:36.330] SensorUtils.computeAltitude(103245.5, 101325.0) -> -158.677439

```

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own

the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

Unit tests MGOS Application

This directory contains some small unit tests for the application logic. It is meant to compile and run on the host operating system, in the author's case this is Ubuntu LTS. To run the test suite:

```
make  
./test
```

prometheus-metrics

A Mongoose OS Prometheus Metrics library.

Introduction

[Prometheus](#) is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has

a very active developer and user community. It is now a standalone open source project and maintained independently of any company.

Mongoose OS is a purpose-built secure Operating System for commercial connected devices. It focuses on stable and secure functioning of multiple connected devices in production and post-sale stages. Key features include secure communication (TLS), over-the-air updates (OTA) and remote device management. These features are usually missing from SDK and their correct implementation would be a complex and resource consuming task. Neglecting them may result in compromised device security and negative brand perception of your products.

Structure

`prometheus-metrics` is a library component that can be added to the app's `mos.yml` file without any configuration needed out of the box, and it pulls in the `http-server` module. The library opens a `/metrics` endpoint which exposes the operating system and library vitalsigns to Prometheus.

By adding the library to the build manifest in `mos.yml`, a compiler define `MGOS_HAVE_PROMETHEUS_METRICS` is set, which other libraries can use to create metrics and update them. This is *non intrusive* because if the library is not used, no additional code is compiled in Mongoose OS and its libraries.

Implementation

Base MGOS Metrics

All Mongoose vitals (memory, WiFi/Ethernet, CPU, scheduling) are exposed using the `mgos_` prefix.

```
# HELP mgos_build Build info
<p class="mume-header" id="help-mgos_build-build-info"></p>

# TYPE mgos_build gauge
<p class="mume-header" id="type-mgos_build-gauge"></p>

mgos_build{app="empty",id="20171121-164823????",version="1.1.04"} 1
# HELP mgos_platform Platform information
<p class="mume-header" id="help-mgos_platform-platform-information"></p>

# TYPE mgos_platform gauge
<p class="mume-header" id="type-mgos_platform-gauge"></p>

mgos_platform{arch="esp32",mac="240AC4106560",idf="v1.0-2815-g50a73c1"} 1
# HELP mgos_uptime Uptime in seconds
<p class="mume-header" id="help-mgos_uptime-uptime-in-seconds"></p>

# TYPE mgos_uptime counter
<p class="mume-header" id="type-mgos_uptime-counter"></p>

mgos_uptime 1888
# HELP mgos_heap_size System memory size
<p class="mume-header" id="help-mgos_heap_size-system-memory-size"></p>

# TYPE mgos_heap_size gauge
<p class="mume-header" id="type-mgos_heap_size-gauge"></p>

mgos_heap_size 295076
```

Platform Specific Metrics

Platform specific vitals are exposed using the `$platform_` prefix, for example `esp32_` for ESP32 and ESP-IDF metrics.

```
# HELP esp32_chip_info ESP32 Chip Information
<p class="mume-header" id="help-esp32_chip_info-esp32-chip-information"></p>

# TYPE esp32_chip_info gauge
<p class="mume-header" id="type-esp32_chip_info-gauge"></p>

esp32_chip_info{model=0,cores=2,revision=1,features=32,sdk="master"} 1
# HELP esp32_num_tasks ESP32 FreeRTOS task count
<p class="mume-header" id="help-esp32_num_tasks-esp32-freertos-task-count"></p>

# TYPE esp32_num_tasks gauge
<p class="mume-header" id="type-esp32_num_tasks-gauge"></p>

esp32_num_tasks 9
```

Library Specific Metrics

Library owners gate the code that creates, updates and exposes the metrics by the define `MGOS_HAVE_PROMETHEUS_METRICS`. Metrics should be defined as static variables to stay private to the implementation. Then, a callback function is installed, and `prometheus-metrics` will loop over all registered callbacks to allow them to add their metrics to the output.

Taking `mqtt` as an example:

```
# if MGOS_HAVE_PROMETHEUS_METRICS
<p class="mume-header" id="if-mgos_have_prometheus_metrics"></p>

# include "mgos_prometheus_metrics.h"
<p class="mume-header" id="include-mgos_prometheus_metricsh"></p>

static uint32_t metrics_mqtt_sent_topics_count = 0;
static uint32_t metrics_mqtt_sent_topics_bytes_total = 0;
static uint32_t metrics_mqtt_received_topics_count = 0;
static uint32_t metrics_mqtt_received_topics_bytes_total = 0;

static void metrics_mqtt(struct mg_connection *nc, void *user_data) {
    mgos_prometheus_metrics_printf(nc, COUNTER,
        "mgos_mqtt_sent_topics_count", "MQTT topics sent",
        "%u", metrics_mqtt_sent_topics_count);

    mgos_prometheus_metrics_printf(nc, COUNTER,
        "mgos_mqtt_sent_topics_bytes_total", "Total bytes sent in MQTT topics",
        "%u", metrics_mqtt_sent_topics_bytes_total);

    mgos_prometheus_metrics_printf(nc, COUNTER,
        "mgos_mqtt_received_topics_count", "MQTT topics sent",
        "%u", metrics_mqtt_received_topics_count);

    mgos_prometheus_metrics_printf(nc, COUNTER,
        "mgos_mqtt_received_topics_bytes_total", "Total bytes received in MQTT topics",
        "%u", metrics_mqtt_received_topics_bytes_total);

    (void) user_data;
}
#endif // MGOS_HAVE_PROMETHEUS_METRICS
<p class="mume-header" id="endif-mgos_have_prometheus_metrics"></p>
```

Then in the library's `init` function, register the callback:

```

bool mgos_mqtt_init(void) {
# if MGOS_HAVE_PROMETHEUS_METRICS
<p class="mume-header" id="if-mgos_have_prometheus_metrics-1"></p>

    mgos_prometheus_metrics_add_handler(metrics_mqtt, NULL);
# endif
<p class="mume-header" id="endif-3"></p>

    return true;
}

```

As mentioned above, if the `prometheus-metrics` library is not included in the app's `mos.yml` manifest, no code will be compiled which makes the addition *non intrusive*.

Application Specific Metrics

Users are able to add their app's own metrics in the same way as libraries can. They do this by registering a handler function, which is called from `prometheus-metrics`.

```

#include "mgos_prometheus_metrics.h"
<p class="mume-header" id="include-mgos_prometheus_metricsh-1"></p>

uint32_t my_counter=0;

static void prometheus_metrics_fn(struct mg_connection *nc, void *user_data) {
    mgos_prometheus_metrics_printf(nc, COUNTER,
        "my_counter", "Total things counted",
        "%u", my_counter);
    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_prometheus_metrics_add_handler(prometheus_metrics_fn, NULL);
    return MGOS_APP_INIT_SUCCESS;
}

```

POSTing to Pushgateway

Prometheus offers an intermediate receiver called a `Pushgateway`, see their [codebase](#) for details. Some users may not wish to have their Mongoose IoT device listen on the network for HTTP connections to the `/metrics` endpoint, for security reasons. As an alternative, the library can be configured to close its serving endpoint, and push its metrics upstream instead. In `mos.yml`:

```

config_schema:
  - ["prometheus.server_enable", false]
  - ["prometheus.pushgateway", "s", "example.com:9091"]

```

An example program using a timer to POST every 5 seconds:

```
# include "mgos.h"
<p class="mume-header" id="include-mgosh-20"></p>

# include "mgos_prometheus_metrics.h"
<p class="mume-header" id="include-mgos_prometheus_metricsh-2"></p>

static void timer_cb(void *user_data) {
    mgos_prometheus_metrics_push("test1", "instance1");
    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_set_timer(5000, true, timer_cb, NULL);
    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

FreeRTOS Kernel

This is official FreeRTOS 10.2.0 kernel from www.freertos.org with some irrelevant stuff removed.

It is used by all the ports except ESP32, which uses its own, heavily modified version.

A device provision library

This library provides the following functionality:

Reset to factory defaults on button press

This functionality allows to reset the device to factory defaults by performing the following steps:

- remove `conf9.json` which holds user-specific device configuraiton
- perform filesystem garbage collection
- reboot the device

Reset to factory defaults is done by configuring a reset button, attached to a certain GPIO pin. Two modes are possible:

- Hold the reset button and reboot the device while holding a button.
For this, set `provision.btn.pin` to a non-negative value, and set `provision.btn.hold_ms` to 0.
- Just hold the reset button pressed for a long time.
For this, set `provision.btn.pin` to a non-negative value, and set `provision.btn.hold_ms` to a positive value, e.g. 5000 - 5 seconds.
If you also set `provision.btn.inhibit_after_s` value to non-zero number of seconds, reset button will be inhibited once uptime reaches this value (reset button will stop working N seconds after boot).

Reset to factory defaults on bad WiFi credentials

This functionality resets the device to factory defaults if the WiFi configuration is entered by user incorrectly (wrong network name or wrong password).

If the device has at least once connected to the WiFi station successfully, reset is never done afterwards.

This is done via configuration parameter `provision.wifi_configured`, which is set to `false` on a fresh device. When the device first connects to the WiFi station, `provision.wifi_configured` is set to `true`. When WiFi connection fails, and `provision.wifi_configured` is `false`, factory reset is triggered.

Provisioning state

Device's provisioning state is tracked as two variables: current and maximum.

Both start at 0 but the current state resets on each boot and maximum state is persisted across reboots.

Maximum state can be used to ensure successful provisioning by specifying a "stable state" (`provision.stable_state`) and a timeout (`provision.timeout`).

If stable state is not reached within the specified time, device config is wiped and device is rebooted (which also resets max state to 0).

Four states are pre-defined:

- Unprovisioned (0). Nothing is configured, will stay in this state without reboot no matter what stable state is configured.
- Networking configured, connecting (1)
- Connecting to cloud (2)
- Connected to cloud (3)

Default value for `provision.stable_state` is 3, so the device will assume it has been fully provisioned once connected to the cloud.

If cloud connection is not required, this can be lowered to 2. If set to 0, device reset is not performed.

Stable state higher than 3 can be used if additional provisioning steps after successful connection are required.

User application code will need to manage further provisioning and use `mgos_provision_set_cur_state()` to signal state transitions.

When state transition occurs an event is raised which application code can use to indicate current provisioning state.

LED indication

If `provision.led.pin` is configured, an event handler will be installed that will use LED to visually indicate the current provisioning state:

- Unprovisioned (0): Blink the LED slowly (once a second)
- Networking configured, connecting (1): Blink faster (every 0.5s)
- Connecting to cloud (2): Blink fast (every 0.25s)
- Connected to cloud (3, or whatever configured stable state is): Solid on

Configuration parameters reference

```

"provision": {
  "configured": false, // Set to true when first time connected to WiFi
  "button": {
    "pin": 17,          // Reset button GPIO number. Negative value disables reset
                      // button functionality. Default: arch-specific.
    "hold_ms": 0,       // Number of milliseconds to hold to trigger a factory reset.
                      // If negative, disable. If 0, reset on boot.
                      // If positive, reset when pressed for that many milliseconds.
  }
}

```

Hello JavaScript/mJS

Intro

This example shows how to use the library with JavaScript or rather mJS.

Prerequisite

Setup [Mongoose OS](https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md) as described here: <https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md>

Please note that the chosen configuration is for a [Adafruit HUZZAH ESP32](#) or a [Adafruit HUZZAH ESP8266](#), and a [\(monochrome\) ePaper Display FeatherWing](#). So please adjust the configuration (GPIOs, display width/height/driver, etc. in the config_schema section) in `mos.yml` accordingly for your setup.

Installation

```

git checkout https://github.com/bbilger/mgos-arduino-adafruit-epd
cd mgos-arduino-adafruit-epd/examples/hello_js

mos build --local --platform esp32 && mos flash && mos console
# or
<p class="mume-header" id="or"></p>

mos build --local --platform esp8266 && mos flash && mos console

```

Hello C++

Intro

This example shows how to use the library with C++.

Prerequisite

Setup [Mongoose OS](https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md) as described here: <https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md>

Please note that the chosen configuration is for a [Adafruit HUZZAH ESP32](#) or a [Adafruit HUZZAH ESP8266](#), and a [\(monochrome\) ePaper Display FeatherWing](#). So please adjust the configuration (GPIOs, display width/height/driver, etc. in the config_schema section) in `mos.yml` accordingly for your setup.

Installation

```
git checkout https://github.com/bbilger/mgos-arduino-adafruit-epd
cd mgos-arduino-adafruit-epd/examples/hello_cpp

mos build --local --platform esp32 && mos flash && mos console
# or
<p class="mume-header" id="or-1"></p>

mos build --local --platform esp8266 && mos flash && mos console
```

Hello C

Intro

This example shows how to use the library with C.

Prerequisite

Setup [Mongoose OS](https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md) as described here: <https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md>

Please note that the chosen configuration is for a [Adafruit HUZZAH ESP32](#) or a [Adafruit HUZZAH ESP8266](#), and a [\(monochrome\) ePaper Display FeatherWing](#). So please adjust the configuration (GPIOs, display width/height/driver, etc. in the config_schema section) in `mos.yml` accordingly for your setup.

Installation

```
git checkout https://github.com/bbilger/mgos-arduino-adafruit-epd
cd mgos-arduino-adafruit-epd/examples/hello_c

mos build --local --platform esp32 && mos flash && mos console
# or
<p class="mume-header" id="or-2"></p>

mos build --local --platform esp8266 && mos flash && mos console
```

Adafruit E-Paper / E-Ink Library for Mongoose OS

Introduction

This is a port of or rather a wrapper around the [Adafruit EPD Library](#) for [Mongoose OS](#).

Please check [Adafruit EPD Library](#) for additional information
and also please note the original [README](#).

Notes:

- this was moved from [bbilger/mgos-arduino-adafruit-epd](#) which was forked from [adafruit/Adafruit_EPD](#)
- EPD stands for e-paper display.

Usage

Examples

There are three examples that show how to use this library:

- [C usage example](#)
- [JavaScript or rather mJS usage example](#)
- [C++ usage example](#)

C

```
# include "mgos_aepd.h"
<p class="mume-header" id="include-mgos_aepdh-2"></p>

...
// get the e-paper display as configured in mos.yml
Adafruit_EPD *epd = mgos_aepd_get_global();
// methods of the Adafruit_EPD C++ class
// can be invoked through "mgos_aepd_<snake_case_method_name>(epd, ...)"
mgos_aepd_some_method(epd, ...);
...
```

JavaScript / mJS

```
load('api_arduino_adafruit_epd.js');
// get the e-paper display as configured in mos.yml
let epd = ArduinoAdafruitEpd.createGlobal();
// the JavaScript object has the same / most methods the original Adafruit_EPD C++ class has
epd.someMethod(...);
```

C++

```
# include "Adafruit_EPD.h"
<p class="mume-header" id="include-adafruit_epdh-1"></p>

# include "mgos_aepd.h"
<p class="mume-header" id="include-mgos_aepdh-3"></p>

...
// get the e-paper display as configured in mos.yml
Adafruit_EPD *epd = mgos_aepd_get_global();
// nothing fancy here, since one can use the original Adafruit_EPD C++ class directly
epd.someMethod(...);
```

Configuration

It's optional to setup the display using the following config but unless you are using more than one display this is probably what you want.

config	type	default	required	comment
aepd.enable	bool	false	-	If enabled, then the display will be setup automatically by the library and the rest of the config must be valid.
aepd.begin	bool	true	-	Calls <code>begin</code> on the display automatically on start.
aepd.driver	string	empty	true	The driver to use for the connected display. At the moment the following drivers are implemented: "IL0373", "IL0398", "IL91874", "SSD1608", "SSD1675".

aepd.width	int	-1	true	Display width in pixels.
aepd.height	int	-1	true	Display height in pixels.
aepd.epd_spi_cs_index	int	-1	true	spi.csX_gpio index for the EDP, 0, 1 or 2.
aepd.sram_spi_cs_index	int	-1	false	spi.csX_gpio index for the SRAM, 0, 1 or 2, or -1 to not use SRAM.
aepd.epd_dc_gpio	int	-1	true	EPD DC GPIO.
aepd.epd_reset_gpio	int	-1	false	EPD reset GPIO or -1
aepd.debug	bool	false	-	At the moment it only prevents a bootloop on misconfiguration.

If you whish to not setup the display via configuration, then you can set it up in C via `mgos_aepd_create`, in C++ by directly instantiating a concrete subclass of `Adafruit_EPD` (e.g. `Adafruit_SSD1675`), and in JS you are on your own but you can for example "ffi" `mgos_aepd_create`.

License

This library is - like the original one - licensed under [MIT license](#).

Adafruit EPD Library

[build passing](#)

This is a library for the Adafruit E-paper displays:

- <https://www.adafruit.com/products/3625>

Check out the links above for our tutorials and wiring diagrams. These devices use SPI to communicate

Adafruit invests time and resources providing this open source code, please support Adafruit and open-source hardware by purchasing products from Adafruit!

Written by Dean Miller for Adafruit Industries.

MIT license, all text above must be included in any redistribution

Implementation of Mongoose OS OTA HTTP server

Overview

This library adds several HTTP endpoints that allow to `POST` new firmware to the device over HTTP:

- `/update` - accept new firmware uploads via HTTP `POST`.
- `/update/revert` - roll back to the previous firmware.
- `/update/commit` - commit new firmware.

Example using `curl` tool (use Mac/Linux terminal or Windows command prompt).

Assume you have build a new firmware for your app. The zip file with a built firmware is located at `build/fw.zip`. In order to update a live device with IP address `IP_ADDRESS`, do:

```
$ curl -i -F filedata=@./build/fw.zip http://IP_ADDRESS/update
```

HTTP/1.1 200 OK

Server: Mongoose/6.10

Content-Type: text/plain

Connection: close

Update applied, finalizing

IR protocol library

Usage

Either

src/main.c

```
# include "mgos.h"
<p class="mume-header " id="include-mgosh-21"></p>

# include "mgos_ir.h"
<p class="mume-header " id="include-mgos_irh"></p>

static void irrecv_cb(int code, void *arg)
{
    LOG(LL_INFO, ("IR: %08X", code));
    (void) arg;
}

enum mgos_app_init_result mgos_app_init(void)
{
    // TSOP on pin 14, NEC protocol
    mgos_irrecv_nec_create(14, irrecv_cb, NULL);

    return MGOS_APP_INIT_SUCCESS;
}
```

or

fs/init.js

```
load("api_ir.js");

// TSOP on pin 14, NEC protocol
let ir = IR.Receiver.NEC.create(14, function(code) {
    print("IR", code);
}, null);
```

MAX72xx LED Matrix Display Library

IMPORTANT NOTE: Please make sure that you find and read the html documentation that comes with the library (open "docs/index.html") or use the link below. **You will need to edit the MAX72xx.h file to configure the type of matrix you are using.** This is the most asked support question so avoid frustration and READ THE MANUAL in the docs subfolder.

The library implements functions that allow the MAX72xx to be used for LED matrices (64 individual LEDs), allowing the programmer to use the LED matrix as a pixel device, displaying graphics elements much like any other pixel addressable display.

In this scenario, it is convenient to abstract out the concept of the hardware device and create a uniform and consistent pixel address

space, with the libraries determining device and device-element address. Similarly, control of the devices is uniform and abstracted to a system level.

The library still retains flexibility for device level control, should the developer require, through the use of overloaded class methods.

[Library Documentation](#)

RPC Service - OTA (Over The Air updates)

This service provides an ability to manage OTA on devices remotely.

It is possible to call this service programmatically via serial, HTTP/RESTful,

Websocket, MQTT or other transports

(see [RPC section](#)) or via the `mos` tool.

See in-depth description of our OTA mechanism at

[Updating firmware reliably - embedded.com](#).

See OTA video tutorial:

Below is a list of exported RPC methods and arguments:

OTA.Update

Trigger OTA firmware update. Arguments:

```
{  
  "url": "https://foo.com/fw123.zip", // Required. URL to the new firmware.  
  "commit_timeout": "300"           // Optional. Time frame in seconds to do  
                                // OTA.Commit after reboot. If commit is  
                                // not done during the timeout, OTA rolls back.  
}
```

A new firmware gets downloaded to the separate flash partition,
and is marked dirty. When the download is complete, device is rebooted.

After reboot, a firmware partition could become committed by calling
`OTA.Commit` - in which case, it is marked as "good". Otherwise, a device
reboots back into the old firmware after the `commit_timeout` seconds.

Example usage:

```
mos call OTA.Update '{"url": "http://1.2.3.4/fw.zip", "commit_timeout": 300}'
```

OTA.Commit

Commit current firmware. Arguments: none.

Example usage:

```
mos call OTA.Commit
```

OTA.Revert

Rolls back to the previous firmware. Arguments: none.

Example usage:

```
mos call OTA.Revert
```

OTA.CreateSnapshot

Create new firmware partition with the copy of currently running firmware. Arguments:

```
{
    // Optional. If true, then current firmware is uncommitted, and needs to
    // be explicitly committed after the first reboot. Otherwise, it'll reboot
    // into the created snapshot. This option is useful if a dangerous, risky
    // live update is to be done on the living device. Then, if the update
    // fails and device bricks, it'll revert to the created good snapshot.
    "set_as_revert": false,
    // Optional. Same meaning as for OTA.Update
    "commit_timeout": "300"
}
```

Example usage:

```
mos call OTA.CreateSnapshot
```

OTA.GetBootState

Get current boot state. Arguments: none.

Example usage:

```
mos call OTA.GetBootState
{
    "active_slot": 0,      # Currently active flash partition.
    "is_committed": true, # Current firmware is marked as "good" (committed).
    "revert_slot": 0,      # If uncommitted, slot to roll back to.
    "commit_timeout": 0   # Commit timeout.
}
```

OTA.SetBootState

Get current boot state. Arguments: see [OTA.GetBootState](#) reply section.

Example usage:

```
mos call OTA.SetBootState '{"revert_slot": 1}'
```

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

Mongoose OS Barometer Library

Introduction

This library provides a simple API that describes barometer sensors. It

implements various popular I2C and SPI barometers behind that API. Callers of the library can rely on the API returning consistently typed data regardless of the choice of sensor.

Primitives

(work in progress)

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

System Configuration over Bluetooth GATT Service

Overview

This library provides a way to examine and change configuration over Generic Attribute Service (GATT) Bluetooth Low-Energy (BLE) service.

The service is designed to be usable with any generic BLE mobile app that supports GATT, e.g. BLE Scanner ([Android](#), [iOS](#)).

Note: Default BT configuration is permissive. See <https://github.com/mongoose-os-libs/bt-common#security> for a better idea.

Attribute description

The service UUID is `5f6d4f53-5f43-4647-5f53-56435f49445f`, which is a representation of a 16-byte string `_mOS_CFG_SVC_ID_`.

The service defines 3 characteristics (attributes):

- `306d4f53-5f43-4647-5f6b-65795f5f5f30 (0mOS_CFG_key__0)` - a write-only attribute that selects the configuration key to be operated on. The key is a string of components separated by period, e.g. `wifi.sta.ssid` or `debug.level`.
- `316d4f53-5f43-4647-5f76-616c75655f31 (1mOS_CFG_value_1)` - a read-write attribute that returns value of the selected key when read or accepts the value to be set. Note that value is not applied immediately after writing. Instead, consecutive writes to this key are appended to form the value to be set (this is due to small value of the default MTU and differences in how clients handle writes exceeding the MTU). For easy manual entry, attribute values are returned as strings and expected as strings as well. So, to enter value of 123, one should submit string `123`, not hex value `0x7b`. Boolean values should be entered as strings `true` or `false`.
- `326d4f53-5f43-4647-5f73-6176655f5f32 (2mOS_CFG_save_2)` - a write-only attribute that applies value submitted to the value attribute to the key selected via the key attribute. The value written to this key can be one of:
 - 0 - just set the value, change will not be persisted
 - 1 - set and save the config
 - 2 - set, save and reboot

Example - configuring WiFi

Here is an example of the configuration provisioning a device - configuring WiFi station settings (we'll use `key`, `value` and `save` to abbreviate long UUIDs):

- Set the SSID, do not save the config yet
 - `wifi.sta.ssid` -> `key`
 - `MyNetwork` -> `value`

- 0 -> save
- Set the password, do not save the config yet
 - wifi.sta.pass -> key
 - MyPassword -> value
 - 0 -> save
- Enable the station, save and reboot
 - wifi.sta.enable -> key
 - true -> value
 - 2 -> save

Once device is confirmed to have successfully connected to WiFi, BT config can be disabled:

- bt.config_enable -> key # Or bt.enable to disable Bluetooth entirely.
- false -> value
- 2 -> save # Save and reboot with BT configuration disabled.

See Also

See [rpc-gatts](#) library,

which provides a GATT service that acts as an RPC channel. It accepts incoming RPC frames and can send them as well - or rather, makes them available for collection.

BME680 sensor API

Introduction

This package contains the Bosch Sensortec's BME680 gas sensor API

The sensor driver package includes bme680.h, bme680.c and bme680_defs.h files

Version

File	Version	Date
bme680.c	3.5.8	22 Feb 2018
bme680.h	3.5.8	22 Feb 2018
bme680_defs.h	3.5.8	22 Feb 2018

Integration details

- Integrate bme680.h, bme680_defs.h and bme680.c file in to your project.
- Include the bme680.h file in your code like below.

```
# include "bme680.h"
<p class="mume-header" id="include-bme680h"></p>
```

File information

- bme680_defs.h : This header file has the constants, macros and datatype declarations.

- bme680.h : This header file contains the declarations of the sensor driver APIs.
- bme680.c : This source file contains the definitions of the sensor driver APIs.

Supported sensor interfaces

- SPI 4-wire
- I2C

Usage guide

Initializing the sensor

To initialize the sensor, you will first need to create a device structure. You can do this by creating an instance of the structure bme680_dev. Then go on to fill in the various parameters as shown below

Example for SPI 4-Wire

```
struct bme680_dev gas_sensor;

/* You may assign a chip select identifier to be handled later */
gas_sensor.dev_id = 0;
gas_sensor.intf = BME680_SPI_INTF;
gas_sensor.read = user_spi_read;
gas_sensor.write = user_spi_write;
gas_sensor.delay_ms = user_delay_ms;
/* amb_temp can be set to 25 prior to configuring the gas sensor
 * or by performing a few temperature readings without operating the gas sensor.
 */
gas_sensor.amb_temp = 25;

int8_t rslt = BME680_OK;
rslt = bme680_init(&gas_sensor);
```

Example for I2C

```
struct bme680_dev gas_sensor;

gas_sensor.dev_id = BME680_I2C_ADDR_PRIMARY;
gas_sensor.intf = BME680_I2C_INTF;
gas_sensor.read = user_i2c_read;
gas_sensor.write = user_i2c_write;
gas_sensor.delay_ms = user_delay_ms;
/* amb_temp can be set to 25 prior to configuring the gas sensor
 * or by performing a few temperature readings without operating the gas sensor.
 */
gas_sensor.amb_temp = 25;

int8_t rslt = BME680_OK;
rslt = bme680_init(&gas_sensor);
```

Regarding compensation functions for temperature, pressure, humidity and gas we have two implementations.

- Integer version
- floating point version

By default, Integer version is used in the API

If the user needs the floating point version, the user has to un-comment BME680_FLOAT_POINT_COMPENSATION macro in bme680_defs.h file or to add it in the compiler flags.

Configuring the sensor

Example for configuring the sensor in forced mode

```
uint8_t set_required_settings;

/* Set the temperature, pressure and humidity settings */
gas_sensor.tph_sett.os_hum = BME680_OS_2X;
gas_sensor.tph_sett.os_pres = BME680_OS_4X;
gas_sensor.tph_sett.os_temp = BME680_OS_8X;
gas_sensor.tph_sett.filter = BME680_FILTER_SIZE_3;

/* Set the remaining gas sensor settings and link the heating profile */
gas_sensor.gas_sett.run_gas = BME680_ENABLE_GAS_MEAS;
/* Create a ramp heat waveform in 3 steps */
gas_sensor.gas_sett.heatr_temp = 320; /* degree Celsius */
gas_sensor.gas_sett.heatr_dur = 150; /* milliseconds */

/* Select the power mode */
/* Must be set before writing the sensor configuration */
gas_sensor.power_mode = BME680_FORCED_MODE;

/* Set the required sensor settings needed */
set_required_settings = BME680_OST_SEL | BME680_OSP_SEL | BME680_OSH_SEL | BME680_FILTER_SEL
| BME680_GAS_SENSOR_SEL;

/* Set the desired sensor configuration */
rslt = bme680_set_sensor_settings(set_required_settings, &gas_sensor);

/* Set the power mode */
rslt = bme680_set_sensor_mode(&gas_sensor);

/* Get the total measurement duration so as to sleep or wait till the
 * measurement is complete */
uint16_t meas_period;
bme680_get_profile_dur(&meas_period, &gas_sensor);
user_delay_ms(meas_period); /* Delay till the measurement is ready */
```

Reading sensor data

Example for reading all sensor data

```
struct bme680_field_data data;

while(1)
{
    rslt = bme680_get_sensor_data(&data, &gas_sensor);

    printf("T: %.2f degC, P: %.2f hPa, H %.2f %%rH ", data.temperature / 100.0f,
           data.pressure / 100.0f, data.humidity / 1000.0f );
    /* Avoid using measurements from an unstable heating setup */
    if(data.status & BME680_GASM_VALID_MSK)
        printf(", G: %d ohms", data.gas_resistance);

    printf("\r\n");
}
```

Templates for function pointers

```
void user_delay_ms(uint32_t period)
{
    /*
     * Return control or wait,
     * for a period amount of milliseconds
     */
}
```

```

int8_t user_spi_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     * to be set low to activate the relevant device on the SPI bus
     */

    /*
     * Data on the bus should be like
     * |-----+-----+-----|
     * | MOSI      | MISO      | Chip Select |
     * |-----+-----+-----|
     * | (don't care) | (don't care) | HIGH |
     * | (reg_addr)   | (don't care) | LOW  |
     * | (don't care) | (reg_data[0]) | LOW  |
     * | (...)       | (...)       | LOW  |
     * | (don't care) | (reg_data[len - 1]) | LOW |
     * | (don't care) | (don't care) | HIGH |
     * |-----+-----+-----|
     */

    return rslt;
}

int8_t user_spi_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     * to be set low to activate the relevant device on the SPI bus
     */

    /*
     * Data on the bus should be like
     * |-----+-----+-----|
     * | MOSI      | MISO      | Chip Select |
     * |-----+-----+-----|
     * | (don't care) | (don't care) | HIGH |
     * | (reg_addr)   | (don't care) | LOW  |
     * | (reg_data[0]) | (don't care) | LOW  |
     * | (...)       | (...)       | LOW  |
     * | (reg_data[len - 1]) | (don't care) | LOW |
     * | (don't care) | (don't care) | HIGH |
     * |-----+-----+-----|
     */

    return rslt;
}

int8_t user_i2c_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to store the I2C address of the device
     */

    /*
     * Data on the bus should be like
     * |-----+-----|
     * | I2C action | Data |
     * |-----+-----|
     * | Start      | -   |
     * | Write      | (reg_addr) |
     * | Stop       | -   |
     * | Start      | -   |
     * | Read       | (reg_data[0]) |
     * | Read       | (...)   |
     * | Read       | (reg_data[len - 1]) |
     */

```

```

* | Stop      | -
* |-----+-----| |
*/
```

```

    return rsht;
}
```

```

int8_t user_i2c_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rsht = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to store the I2C address of the device
     */

    /*
     * Data on the bus should be like
     * |-----+-----|
     * | I2C action | Data
     * |-----+-----|
     * | Start      | -
     * | Write      | (reg_addr)
     * | Write      | (reg_data[0])
     * | Write      | (....)
     * | Write      | (reg_data[len - 1])
     * | Stop       | -
     * |-----+-----|
     */

```

```

    return rsht;
}

```

Copyright (C) 2017 - 2018 Bosch Sensortec GmbH# Change Log

All notable changes to the BME680 Sensor API will be documented in this file.

v3.5.3, 30 Oct 2017

Changed

- Changed the compensation equation formulae to use shifting operation
- Updated the "bme680_get_profile_dur" API
- Fixed Checkpatch and made linux compatible

v3.5.2, 18 Oct 2017

Changed

- Fixed bug of temperature compensation in pressure

v3.5.1, 5 Jul 2017

Changed

- Fixed bug with overwriting of the result with communication results
- Added member in the dev structure to store communication results
- Updated set profile duration API to not return a result.
- Added new API to get the duration for the existing profile
- Fixed bug with setting gas configuration. Reduced to writing only relevant bytes

- Updated readme
- Updated documentation for the type definitions
- Removed mode check for get sensor data and setting and getting profile dur

v3.5.0, 28 Jun 2017

Changed

- Fixed bug with getting and setting mem pages
- Changed initialization sequence to be more robust
- Added additional tries while reading data in case of inadequate delay

v3.4.0, 8 Jun 2017

Changed

- Modified the bme680_get_sensor_data API. User has to now pass the struct that stores the data rather than retrieving from the bme680_dev structure.
- Fixed possible bugs

v3.3.0, 24 May 2017

Changed

- Name changes in the BME680 device structure.
- Removed sequential and parallel modes.
- Removed ODR related sensor settings
- Modified get sensor settings API with user selection.
- Removed sort sensor data and swap fields API which are not required.

Added

- BME680 set profile duration API.

v3.2.1, 17 May 2017

Added

- Took the reference as base version 3.2.1 of BME680 sensor and added.

BME680 sensor API

Introduction

This package contains the Bosch Sensortec's BME680 gas sensor API

The sensor driver package includes bme680.h, bme680.c and bme680_defs.h files

Version

File	Version	Date
bme680.c	3.5.9	19 Jun 2018
bme680.h	3.5.9	19 Jun 2018
bme680_defs.h	3.5.9	19 Jun 2018

Integration details

- Integrate bme680.h, bme680_defs.h and bme680.c file in to your project.
- Include the bme680.h file in your code like below.

```
# include "bme680.h"
<p class="mume-header" id="include-bme680h-1"></p>
```

File information

- bme680_defs.h : This header file has the constants, macros and datatype declarations.
- bme680.h : This header file contains the declarations of the sensor driver APIs.
- bme680.c : This source file contains the definitions of the sensor driver APIs.

Supported sensor interfaces

- SPI 4-wire
- I2C

Usage guide

Initializing the sensor

To initialize the sensor, you will first need to create a device structure. You can do this by creating an instance of the structure bme680_dev. Then go on to fill in the various parameters as shown below

Example for SPI 4-Wire

```
struct bme680_dev gas_sensor;

/* You may assign a chip select identifier to be handled later */
gas_sensor.dev_id = 0;
gas_sensor.intf = BME680_SPI_INTF;
gas_sensor.read = user_spi_read;
gas_sensor.write = user_spi_write;
gas_sensor.delay_ms = user_delay_ms;
/* amb_temp can be set to 25 prior to configuring the gas sensor
 * or by performing a few temperature readings without operating the gas sensor.
 */
gas_sensor.amb_temp = 25;

int8_t rslt = BME680_OK;
rslt = bme680_init(&gas_sensor);
```

Example for I2C

```

struct bme680_dev gas_sensor;

gas_sensor.dev_id = BME680_I2C_ADDR_PRIMARY;
gas_sensor.intf = BME680_I2C_INTF;
gas_sensor.read = user_i2c_read;
gas_sensor.write = user_i2c_write;
gas_sensor.delay_ms = user_delay_ms;
/* amb_temp can be set to 25 prior to configuring the gas sensor
 * or by performing a few temperature readings without operating the gas sensor.
 */
gas_sensor.amb_temp = 25;

int8_t rslt = BME680_OK;
rslt = bme680_init(&gas_sensor);

```

Regarding compensation functions for temperature, pressure, humidity and gas we have two implementations.

- Integer version
- floating point version

By default, Integer version is used in the API

If the user needs the floating point version, the user has to un-comment BME680_FLOAT_POINT_COMPENSATION macro in bme680_defs.h file or to add it in the compiler flags.

Configuring the sensor

Example for configuring the sensor in forced mode

```

uint8_t set_required_settings;

/* Set the temperature, pressure and humidity settings */
gas_sensor.tph_sett.os_hum = BME680_OS_2X;
gas_sensor.tph_sett.os_pres = BME680_OS_4X;
gas_sensor.tph_sett.os_temp = BME680_OS_8X;
gas_sensor.tph_sett.filter = BME680_FILTER_SIZE_3;

/* Set the remaining gas sensor settings and link the heating profile */
gas_sensor.gas_sett.run_gas = BME680_ENABLE_GAS_MEAS;
/* Create a ramp heat waveform in 3 steps */
gas_sensor.gas_sett.heatr_temp = 320; /* degree Celsius */
gas_sensor.gas_sett.heatr_dur = 150; /* milliseconds */

/* Select the power mode */
/* Must be set before writing the sensor configuration */
gas_sensor.power_mode = BME680_FORCED_MODE;

/* Set the required sensor settings needed */
set_required_settings = BME680_OST_SEL | BME680_OSP_SEL | BME680_OSH_SEL | BME680_FILTER_SEL
| BME680_GAS_SENSOR_SEL;

/* Set the desired sensor configuration */
rslt = bme680_set_sensor_settings(set_required_settings,&gas_sensor);

/* Set the power mode */
rslt = bme680_set_sensor_mode(&gas_sensor);

```

Reading sensor data

Example for reading all sensor data

```

/* Get the total measurement duration so as to sleep or wait till the
 * measurement is complete */
uint16_t meas_period;
bme680_get_profile_dur(&meas_period, &gas_sensor);

struct bme680_field_data data;

while(1)
{
    user_delay_ms(meas_period); /* Delay till the measurement is ready */

    rslt = bme680_get_sensor_data(&data, &gas_sensor);

    printf("T: %.2f degC, P: %.2f hPa, H %.2f %%rH ", data.temperature / 100.0f,
          data.pressure / 100.0f, data.humidity / 1000.0f );
    /* Avoid using measurements from an unstable heating setup */
    if(data.status & BME680_GASM_VALID_MSK)
        printf(", G: %d ohms", data.gas_resistance);

    printf("\r\n");

    /* Trigger the next measurement if you would like to read data out continuously */
    if (gas_sensor.power_mode == BME680_FORCED_MODE) {
        rslt = bme680_set_sensor_mode(&gas_sensor);
    }
}

```

Templates for function pointers

```

void user_delay_ms(uint32_t period)
{
    /*
     * Return control or wait,
     * for a period amount of milliseconds
     */
}

int8_t user_spi_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     * to be set low to activate the relevant device on the SPI bus
     */

    /*
     * Data on the bus should be like
     * |-----+-----+-----|
     * | MOSI      | MISO       | Chip Select |
     * |-----+-----+-----|
     * | (don't care) | (don't care) | HIGH      |
     * | (reg_addr)   | (don't care) | LOW       |
     * | (don't care) | (reg_data[0]) | LOW       |
     * | (....)       | (....)      | LOW       |
     * | (don't care) | (reg_data[len - 1]) | LOW      |
     * | (don't care) | (don't care) | HIGH      |
     * |-----+-----+-----|
     */

    return rslt;
}

int8_t user_spi_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     */

```

```

* to be set low to activate the relevant device on the SPI bus
*/
/*
 * Data on the bus should be like
 * |-----+-----+-----|
 * | MOSI          | MISO          | Chip Select |
 * |-----+-----+-----|
 * | (don't care)   | (don't care)   | HIGH        |
 * | (reg_addr)     | (don't care)   | LOW         |
 * | (reg_data[0])   | (don't care)   | LOW         |
 * | (....)         | (....)         | LOW         |
 * | (reg_data[len - 1]) | (don't care) | LOW         |
 * | (don't care)   | (don't care)   | HIGH        |
 * |-----+-----+-----|
 */
return rslt;
}

int8_t user_i2c_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to store the I2C address of the device
     */
    /*
     * Data on the bus should be like
     * |-----+-----|
     * | I2C action | Data
     * |-----+-----|
     * | Start      | -
     * | Write      | (reg_addr)
     * | Stop       | -
     * | Start      | -
     * | Read       | (reg_data[0])
     * | Read       | (....)
     * | Read       | (reg_data[len - 1])
     * | Stop       | -
     * |-----+-----|
     */
    return rslt;
}

int8_t user_i2c_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to store the I2C address of the device
     */
    /*
     * Data on the bus should be like
     * |-----+-----|
     * | I2C action | Data
     * |-----+-----|
     * | Start      | -
     * | Write      | (reg_addr)
     * | Write      | (reg_data[0])
     * | Write      | (....)
     * | Write      | (reg_data[len - 1])
     * | Stop       | -
     * |-----+-----|
     */
    return rslt;
}

```

Copyright (C) 2017 - 2018 Bosch Sensortec GmbH Parts of the library are covered by different licenses

BME680 driver

Copyright (C) 2017 - 2018 Bosch Sensortec GmbH

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of the copyright holder nor the names of the contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE

The information provided is believed to be accurate and reliable. The copyright holder assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of the copyright holder.

BSEC library

```
/*
 * Copyright (C) 2015, 2016, 2017 Robert Bosch. All Rights Reserved..
 *
 * Disclaimer
 *
 * Common:
 * Bosch Sensortec products are developed for the consumer goods industry. They may only be used
 * within the parameters of the respective valid product data sheet. Bosch Sensortec products are
 * provided with the express understanding that there is no warranty of fitness for a particular purpose.
 * They are not fit for use in life-sustaining, safety or security sensitive systems or any system or device
 * that may lead to bodily harm or property damage if the system or device malfunctions. In addition,
 * Bosch Sensortec products are not fit for use in products which interact with motor vehicle systems.
 * The resale and/or use of products are at the purchasers own risk and his own responsibility. The
 * examination of fitness for the intended use is the sole responsibility of the Purchaser.
 *
 * The purchaser shall indemnify Bosch Sensortec from all third party claims, including any claims for
 * incidental, or consequential damages, arising from any product use not covered by the parameters of
 * the respective valid product data sheet or not approved by Bosch Sensortec and reimburse Bosch
 * Sensortec for all costs in connection with such claims.
 *
 * The purchaser must monitor the market for the purchased products, particularly with regard to
 * product safety and inform Bosch Sensortec without delay of all security relevant incidents.
 *
 * Engineering Samples are marked with an asterisk (*) or (e). Samples may vary from the valid
 * technical specifications of the product series. They are therefore not intended or fit for resale to third
 * parties or for use in end products. Their sole purpose is internal client testing. The testing of an
 * engineering sample may in no way replace the testing of a product series. Bosch Sensortec
 * assumes no liability for the use of engineering samples. By accepting the engineering samples, the
 * Purchaser agrees to indemnify Bosch Sensortec from all claims arising from the use of engineering
 * samples.
 *
 * Special:
 * This software module (hereinafter called "Software") and any information on application-sheets
 * (hereinafter called "Information") is provided free of charge for the sole purpose to support your
 * application work. The Software and Information is subject to the following terms and conditions:
 *
 * The Software is specifically designed for the exclusive use for Bosch Sensortec products by
 * personnel who have special experience and training. Do not use this Software if you do not have the
 * proper experience or training.
 *
 * This Software package is provided `` as is `` and without any expressed or implied warranties,
 * including without limitation, the implied warranties of merchantability and fitness for a particular
 * purpose.
 *
 * Bosch Sensortec and their representatives and agents deny any liability for the functional impairment
 * of this Software in terms of fitness, performance and safety. Bosch Sensortec and their
 * representatives and agents shall not be liable for any direct or indirect damages or injury, except as
 * otherwise stipulated in mandatory applicable law.
 *
 * The Information provided is believed to be accurate and reliable. Bosch Sensortec assumes no
 * responsibility for the consequences of use of such Information nor for any infringement of patents or
 * other rights of third parties which may result from its use. No license is granted by implication or
 * otherwise under any patent or patent rights of Bosch. Specifications mentioned in the Information are
 * subject to change without notice.
 *
 * It is not allowed to deliver the source code of the Software to any third party without permission of
 * Bosch Sensortec.
 *
 */

```

mOS integration code

Copyright (c) 2019 Deomid "rojer" Ryabkov
All rights reserved

Licensed under the Apache License, Version 2.0(the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

```# BME680 sensor driver library for Mongoose OS

## ## Overview

This library integrates the low-level [driver]([https://github.com/BoschSensortec/BME680\\_driver](https://github.com/BoschSensortec/BME680_driver)) and the [BSEC]([https://www.bosch-sensortec.com/bst/products/all\\_products/bsec](https://www.bosch-sensortec.com/bst/products/all_products/bsec)) software library which provides high-level sensor control to obtain reliable air quality sensor data.

This library provides the necessary glue code for both to work under Mongoose OS and a number of helper functions.

When sensor output is ready, an `MGOS\_EV\_BME680\_BSEC\_OUTPUT` event is triggered which receives a structure containing sensor outputs (see `struct mgos\_bme680`).

## ### Platform support

Currently only supported on ESP8266 and ESP32 platforms, ARM support is a 'TODO'.

## ## Quick Start

The library is configured through the `bme680` configuration section, defined [here](mos.yml).

Several things need to be done to obtain readings from the sensor:

- I2C must be enabled, set `i2c.enable=true` (currently only I2C interface is supported).
- `bme680.i2c\_addr` must be set to the correct address. It's either 0x76 or 0x77 depending on the state of the address selection pin.
- `bme680.enable` needs to be set to `true` for library to be initialized at all.

With these and the rest of the settings left in their default state, you should get readings from all the sensors at 3 second interval.

## ## IAQ sensor accuracy

IAQ sensor requires calibration before producing accurate values. Values with accuracy value less than 3 are unreliable.

By default the library will perform calibration automatically (still may take up to 30 minutes to complete).

## ## Configuration details

A number of options are provided for more advanced control of the sensor behavior.

- `bme680.bsec.enable`: normally it is advisable to use the BSEC library to process raw values returned by the sensor. Turning this off will enable you to either use the sensor directly (reference to the dev ice can be obtained via `mgos\_bme680\_get\_global()`) or initialize driver directly.
- `bme680.bsec.config\_file`: BSEC library comes with a number of pre-generated configuration profiles that can be loaded to improve accuracy of the measurements.
- `bme680.bsec.state\_file`, `bme680.bsec.state\_save\_interval`: BSEC library performs estimations over long periods of time and the accuracy of its output relies on these parameters.
- `bme680.bsec.{iaq,temp,rh,ps}\_sample\_rate`: Set sampling rates for different parts of the BME680 multi-sensor. Each can be individually disabled (empty array).
- `bme680.bsec.iaq\_auto\_cal`: if IAQ sensor is enabled (`bme680.bsec.iaq\_sample\_rate` is not empty) and this option is enabled, mos will automatically raise an event when calibration is complete.

## ## Example

With mOS library providing the integration, getting samples from the sensor is very simple - all you need to do is subscribe to the event:

```
```c
#include "mgos.h"
<p class="mume-header" id="include-mgosh-22"></p>

#include "mgos_bme680.h"
<p class="mume-header" id="include-mgos_bme680h"></p>

static void bme680_output_cb(int ev, void *ev_data, void *arg) {
    const struct mgos_bsec_output *out = (struct mgos_bsec_output *) ev_data;
    double ts = out->temp.time_stamp / 1000000000.0;
    float ps_kpa = out->ps.signal / 1000.0f;
    float ps_mmhg = out->ps.signal / 133.322f;
    if (out->iaq.time_stamp > 0) {
        LOG(LL_INFO,
```

```

("%.2f IAQ %.2f (acc %d) T %.2f RH %.2f P %.2f kPa (%.2f mmHg)", ts,
out->iaq.signal, out->iaq.accuracy, out->temp.signal, out->rh.signal,
ps_kpa, ps_mmhg));
} else {
LOG(LL_INFO, ("%.2f T %.2f RH %.2f P %.2f kPa (%.2f mmHg)", ts,
out->temp.signal, out->rh.signal, ps_kpa, ps_mmhg));
}
(void) ev;
(void) arg;
}

enum mgos_app_init_result mgos_app_init(void) {
mgos_event_add_handler(MGOS_EV_BME680_BSEC_OUTPUT, bme680_output_cb, NULL);
return MGOS_APP_INIT_SUCCESS;
}

```

Output:

```

[Aug 26 23:00:59.324] mgos_i2c_gpio_maste:250 I2C GPIO init ok (SDA: 4, SCL: 5, freq: 100000)
[Aug 26 23:00:59.348] mgos_bme680.c:466      BME680 @ 0/0x77 init ok
[Aug 26 23:00:59.353] mgos_bme680.c:396      BSEC 1.4.7.4 initialized
[Aug 26 23:00:59.364] mgos_bme680.c:404      Failed to load BSEC config from bsec_jaq.config: -33, will use defaults
[Aug 26 23:00:59.377] mgos_bme680.c:414      Failed to load BSEC state from bsec.state: -33, will use defaults
...
[Aug 26 23:01:00.337] mgos_init.c:36      Init done, RAM: 51152 total, 41996 free, 42000 min free
[Aug 26 23:01:00.352] mgos_bme680.c:281      IAQ sensor requires calibration
[Aug 26 23:01:00.356] main.c:13      0.68 IAQ 25.00 (acc 0) T 27.07 RH 57.51 P 101.76 kPa (763.28 mmHg)
[Aug 26 23:01:02.641] main.c:13      3.68 IAQ 25.00 (acc 0) T 26.98 RH 57.87 P 101.76 kPa (763.29 mmHg)
[Aug 26 23:01:05.645] main.c:13      6.69 IAQ 25.00 (acc 0) T 26.99 RH 57.88 P 101.76 kPa (763.28 mmHg)
[Aug 26 23:01:08.650] main.c:13      9.69 IAQ 25.00 (acc 0) T 27.01 RH 57.84 P 101.76 kPa (763.28 mmHg)
...
[Aug 26 23:05:54.095] main.c:13      295.11 IAQ 25.00 (acc 0) T 26.72 RH 58.46 P 101.76 kPa (763.27 mmHg)
[Aug 26 23:05:56.868] mgos_bme680.c:368      BSEC state saved (bsec.state)
[Aug 26 23:05:57.100] main.c:13      298.11 IAQ 25.00 (acc 0) T 26.72 RH 58.46 P 101.76 kPa (763.26 mmHg)
...
[Aug 26 23:08:12.326] main.c:13      433.32 IAQ 51.91 (acc 1) T 26.70 RH 58.46 P 101.76 kPa (763.25 mmHg)
[Aug 26 23:08:15.331] main.c:13      436.33 IAQ 250.00 (acc 2) T 26.71 RH 58.41 P 101.76 kPa (763.27 mmHg)
...
[Aug 26 23:16:46.136] main.c:13      947.09 IAQ 49.84 (acc 3) T 26.72 RH 58.90 P 101.76 kPa (763.23 mmHg)
[Aug 26 23:16:49.140] mgos_bme680.c:289      IAQ sensor calibration complete
[Aug 26 23:16:49.144] main.c:13      950.10 IAQ 50.42 (acc 3) T 26.72 RH 58.92 P 101.76 kPa (763.23 mmHg)
[Aug 26 23:16:52.145] main.c:13      953.10 IAQ 50.01 (acc 3) T 26.73 RH 58.88 P 101.76 kPa (763.23 mmHg)
[Aug 26 23:16:55.150] main.c:13      956.10 IAQ 50.95 (acc 3) T 26.73 RH 58.87 P 101.76 kPa (763.23 mmHg)
...

```

License

See [here](#).

A VFS device that encrypts reads and writes

Overview

- AES-128/192/256 are supported (algo: AES-nnn parameter, default is AES-128).
- Reads and writes are encrypted, erases are passed through as is.
- Encryption is performed in ECB mode, key is XORed with offset.
- Reads and writes must be aligned to 16-byte boundaries.
- Writes will be padded to 16 byte block size, so partial writes will only work for last plain-text block.

Hint: If you want an encrypted filesystem, [LFS](#) will work just fine with this method while [SPIFFS](#) will not.

Key source

Key can be supplied directly (as the `key` option) but a better approach is to use a `key device` to obtain the key when required.

Key device can be any other VFS device that supports reads. It can be an existing device (`key_dev: name`) or created in-situ (`key_dev_type + key_dev_opts`).

Hint: To read key from RAM, use the `vfs-dev-ram`.

Hint 2: Want to generate your own key? Create your own VFS device. Don't worry about methods other than `read`.

Example

Options for encrypting `extf0` with AES-256 with key from STM32 OTP area (536836096 = 0x1fff7800).

```
{"dev": "extf0", "algo": "AES-256", "key_dev_type": "RAM", "key_dev_opts": {"addr": 536836096, "size": 32}}
```

Don't forget to add `vfs-dev-ram` to libs.

DNS-SD/Bonjour protocol support library

By default, advertises HTTP server (if enabled).

Additional instances can be added by user's application, example (for Apple HAP):

```
const struct mgos_dns_sd_txt_entry gizmo_txt[] = {
    {.key = "c#", .value = MG_MK_STR("1")},
    {.key = "ff", .value = MG_MK_STR("0")},
    {.key = "pv", .value = MG_MK_STR("1.0")},
    {.key = "id", .value = MG_MK_STR("11:22:33:44:55:66")},
    {.key = "md", .value = MG_MK_STR("Fancy Gizmo 9000")},
    {.key = "s#", .value = MG_MK_STR("1")},
    {.key = "sf", .value = MG_MK_STR("1")},
    {.key = "ci", .value = MG_MK_STR("8")}, // Switch
    {.key = NULL},
};

mgos_dns_sd_add_service_instance("gizmo9000", "_hap._tcp", 8080, gizmo_txt);
```

Cesanta note

Imported from https://github.com/adafruit/Adafruit_BME280_Library/commit/a148538212b07df05881562874171de7f312c81f

This is a library for the Adafruit BME280 Humidity, Barometric Pressure + Temp sensor

Designed specifically to work with the Adafruit BME280 Breakout

- <http://www.adafruit.com/products/2652>

These sensors use I2C or SPI to communicate, up to 4 pins are required to interface

Use of this library also requires [Adafruit_Sensor](#)

to be installed on your local system.

Adafruit invests time and resources providing this open source code,
please support Adafruit and open-source hardware by purchasing
products from Adafruit!

Check out the links above for our tutorials and wiring diagrams

Written by Limor Fried/Ladyada for Adafruit Industries.

BSD license, all text above must be included in any redistribution

To download, click the DOWNLOAD ZIP button, rename the uncompressed folder Adafruit_BME280.

Check that the Adafruit_BME280 folder contains Adafruit_BME280.cpp and Adafruit_BME280.h

Place the Adafruit_BME280 library folder your arduinosketchfolder/libraries/ folder.

You may need to create the libraries subfolder if its your first library. Restart the IDE.

We also have a great tutorial on Arduino library installation at:

<http://learn.adafruit.com/adafruit-all-about-arduino-libraries-install-use>

Compatibility

MCU	Tested Works	Doesn't Work	Not Tested	Notes
Atmega328 @ 16MHz	X			
Atmega328 @ 12MHz	X			
Atmega32u4 @ 16MHz	X			Use SDA/SCL on pins D2 & D3
Atmega32u4 @ 8MHz	X			Use SDA/SCL on pins D2 & D3
ESP8266	X			I2C: just works, SPI: SDA/SCL default to pins 4 & 5 but any two pins can be assigned as SDA/SCL using Wire.begin(SDA,SCL)
ESP32	X			I2C: just works, SPI: SDA/SCL default to pins 4 & 5 but any two pins can be assigned as SDA/SCL using Wire.begin(SDA,SCL)
Atmega2560 @ 16MHz	X			Use SDA/SCL on pins 20 & 21
ATSAM3X8E	X			Use SDA/SCL on pins 20 & 21
ATSAM21D	X			
ATtiny85 @ 16MHz		X		
ATtiny85 @ 8MHz		X		
Intel Curie @ 32MHz			X	
STM32F2			X	

- ATmega328 @ 16MHz : Arduino UNO, Adafruit Pro Trinket 5V, Adafruit Metro 328, Adafruit Metro Mini
- ATmega328 @ 12MHz : Adafruit Pro Trinket 3V
- ATmega32u4 @ 16MHz : Arduino Leonardo, Arduino Micro, Arduino Yun, Teensy 2.0
- ATmega32u4 @ 8MHz : Adafruit Flora, Bluefruit Micro
- ESP8266 : Adafruit Huzzah
- ATmega2560 @ 16MHz : Arduino Mega
- ATSAM3X8E : Arduino Due
- ATSAM21D : Arduino Zero, M0 Pro

- ATtiny85 @ 16MHz : Adafruit Trinket 5V
- ATtiny85 @ 8MHz : Adafruit Gemma, Arduino Gemma, Adafruit Trinket 3V

Arduino Adafruit BME280 and BMP280 library for Mongoose OS

VFS interface for SPIFFS

A collection of libraries for demoing JavaScript code

Overview

This library is intended to be used for apps that implement device logic in JavaScript. It is essentially a collection of libraries that implement a wide set of functionalities - from hardware peripherals API to cloud integrations like AWS IoT, Google IoT Core, etc.

Arduino DallasTemperature library for Mongoose OS

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or

feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.

2. Follow the normal process of [forking](#)

the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.

3. Any significant changes should almost always be accompanied by tests. The

project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.

4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.

5. Do your best to have [well-formed commit messages](#)

for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.

6. Finally, push the commits to your fork and submit a [pull request](#).

Prometheus Sensors

This is a library that implements a set of sensor types, reading their values periodically and exposing them to Prometheus using the `prometheus-metrics` library.

Supported Sensors

DHT

This is using `dht` library; each sensor reading takes approximately 32ms (28ms if there is no sensor on the GPIO in). They are placed on timers so as to minimize pre-emption of the main application. DHT sensors are polled in order based on `sensors.dht_gpio` pins and with a period of `sensors.dht_period` which defaults to 3 seconds.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="DHT"} 3.5
temperature{sensor="1",type="DHT"} 6.2
humidity{sensor="0",type="DHT"} 43.900002
humidity{sensor="1",type="DHT"} 35.599998
```

VEML6075

This is using `veml6075-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x10`, but configurable with `sensors.veml6075_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.veml6075_period` which defaults to 3 seconds, each sensor reading takes approximately 15ms.

Please see the upstream [source](#)

for more information on the driver.

Reported values (all types are gauges):

```
UV{band="UVA",sensor="0",type="VEML6075"} 0  
UV{band="UVB",sensor="0",type="VEML6075"} 2  
UVIndex{sensor="0",type="VEML6075"} 0.00125
```

BME280

This is using `bme280` library; one sensor is allowed based on I2C address on the bus, normally `0x10`, but configurable with `sensors.bme280_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.bme280_period` which defaults to 3 seconds, each sensor reading takes approximately 4ms. Note that there are several versions of this popular barometer chip: BME280 is fully fledged and includes a hygrometer (measuring the relative humidity), while BMP280 does not.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="BME280"} 18.4  
humidity{sensor="0",type="BME280"} 77.4  
pressure{sensor="0",type="BME280"} 96720.4
```

MCP9808

This is using `mcp9808-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x18`, but configurable with `sensors.mcp9808_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.mcp9808_period` which defaults to 3 seconds, each sensor reading takes approximately 3ms.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="MCP9808"} 18.6
```

Si7021

This is using `si7021-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x40`, but configurable with `sensors.si7021_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.si7021_period` which defaults to 3 seconds, each sensor reading takes approximately 54ms.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="SI7021"} 18.6  
humidity{sensor="0",type="SI7021"} 56.4
```

SHT31

This is using `sht31-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x44`, but configurable with `sensors.sht31_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.sht31_period` which defaults to 3 seconds, each sensor reading takes approximately 18ms.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="SHT31"} 18.6
humidity{sensor="0",type="SHT31"} 56.4
```

HTU21D-F

This is using `htu21df-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x40`, but configurable with `sensors.htu21df_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.htu21df_period` which defaults to 3 seconds, each sensor reading takes approximately 105ms.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="HTU21DF"} 18.5
humidity{sensor="0",type="HTU21DF"} 55.8
```

CCS811

This is using `ccs811-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x5A`, but configurable with `sensors.ccs811_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.ccs811_period` which defaults to 3 seconds, each sensor reading takes approximately 5ms.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
eco2{sensor="0",type="CCS811"} 7992
tvoc{sensor="0",type="CCS811"} 1156
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Implementation of Mongoose OS OTA HTTP client

This library adds a device configuration section called `update`, where a device could be configured to poll a specified HTTP URL for a new app firmware.

Also, this library adds a C API to fetch a new firmware from the given URL and update programmatically.

Configuration section

The library adds the following object to the device configuration:

```
"update": {  
    "commit_timeout": 0,           // OTA commit timeout  
    "url": "",                  // HTTP URL to poll  
    "interval": 0,               // Polling interval  
    "extra_http_headers": "",    // Extra HTTP request headers  
    "ssl_ca_file": "ca.pem",     // TLS CA cert file  
    "ssl_client_cert_file": "",  // TLS cert file  
    "ssl_server_name": "",       // TLS server name  
    "enable_post": true  
}
```

Si7005 Sensor Driver for Mongoose OS

Overview

[Si7005](#) is a temperature and relative humidity sensor by Silicon Labs.

This library provides a driver for this device.

API documentation

See `include/si7005.h`.

Example

```
# include "mgos.h"  
<p class="mume-header" id="include-mgosh-23"></p>  
  
# include "si7005.h"  
<p class="mume-header" id="include-si7005h"></p>  
  
static void temp_timer_cb(void *arg) {  
    float temp = si7005_read_temp();  
    float rh = si7005_read_rh();  
    LOG(LL_INFO, ("T %.2f RH %.2f", temp, rh));  
    (void) arg;  
}  
  
enum mgos_app_init_result mgos_app_init(void) {  
    if (si7005_probe()) {  
        LOG(LL_INFO, ("Si7005 sensor found"));  
        mgos_set_timer(1000, MGOS_TIMER_REPEAT, temp_timer_cb, NULL);  
    } else {  
        LOG(LL_WARN, ("Failed to init temp sensor"));  
    }  
    return MGOS_APP_INIT_SUCCESS;  
}
```

Note: You need to make sure that I2C is enabled. This can be achieved by adding

```
config_schema:  
  - ["i2c.enable", true]
```

to mos.yml. You may need to adjust SDA and SCL pins as well.

See [I2C](#) library for details.

How to contribute

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

MCP9808 I2C Driver

A Mongoose library for Microchip's MCP9808 integrated circuit.

Sensor details

Microchip Technology Inc.'s MCP9808 digital temperature sensor converts temperatures between -20 °C and +100 °C to a digital word with $\pm 0.25^\circ\text{C}/\pm 0.5^\circ\text{C}$ (typical/maximum) accuracy.

The MCP9808 comes with user-programmable registers that provide flexibility for temperature sensing applications. The registers allow user-selectable settings such as Shutdown or Low-Power modes and the specification of temperature Alert window limits and critical output limits. When the temperature changes beyond the specified boundary limits, the MCP9808 outputs an Alert signal. The user has the option of setting the Alert output signal polarity as an active-low or activehigh comparator output for thermostat operation, or as a temperature Alert interrupt output for microprocessor based systems. The Alert output can also be configured as a critical temperature output only.

This sensor has an industry standard 400 kHz, 2-wire, SMBus/I2C compatible serial interface, allowing up to eight or sixteen sensors to be controlled with a single serial bus.

These features make the MCP9808 ideal for sophisticated, multi-zone, temperature-monitoring applications.

See [datasheet](#)

for implementation details.

A great place to buy a ready made and tested unit is at [Adafruit](#).

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```

#include "mgos.h"
<p class="mume-header" id="include-mgosh-24"></p>

#include "mgos_i2c.h"
<p class="mume-header" id="include-mgos_i2ch-3"></p>

#include "mgos_mcp9808.h"
<p class="mume-header" id="include-mgos_mcp9808h"></p>

static struct mgos_mcp9808 *s_mcp9808;

static void timer_cb(void *user_data) {
    float temperature;

    temperature=mgos_mcp9808_getTemperature(s_mcp9808);

    LOG(LL_INFO, ("mcp9808 temperature=%.2f", temperature));

    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_i2c *i2c;

    i2c=mgos_i2c_get_global();
    if (!i2c) {
        LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
    } else {
        s_mcp9808=mgos_mcp9808_create(i2c, 0x40); // Default I2C address
        if (s_mcp9808) {
            mgos_set_timer(5000, true, timer_cb, NULL);
        } else {
            LOG(LL_ERROR, ("Could not initialize sensor"));
        }
    }
    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

JSON Store

JSON Store is a library which maintains a JSON file with order-preserving mappings from arbitrary strings to any kind of JSON values (a string, an object, etc). IDs can be either randomly generated or custom.

The order in which items are stored in the file and iterated (with

`mgos_jstore_iterate()`, see `include/mgos_jstore.h`)

is the order of insertion of items to the store.

RPC over GATT, Server

Overview

This library provides a GATT service that acts as an RPC channel.

It accepts incoming frames and can send them as well - or rather, make them available for collection.

Note: Default BT configuration is permissive. See <https://github.com/mongoose-os-libs/bt-common#security> for a better idea.

Attribute description

The service UUID is `5f6d4f53-5f52-5043-5f53-56435f49445f`, which is a representation of a 16-byte string `_mOS_RPC_SVC_ID_`.

Three attributes are defined:

- `5f6d4f53-5f52-5043-5f64-6174615f5f (_mOS_RPC_data_)` - a r/w attribute used to submit frames for tx to the device and read out frames from the device.
- `5f6d4f53-5f52-5043-5f74-785f63746c5f (_mOS_RPC_tx_ctl_)` - a write-only attribute. Before sending a frame expected length of the frame is submitted as a big-endian 32-bit number (so, for a 100 byte frame bytes `00 00 00 64` should be sent), followed by any number of writes to the data attribute. Chunking can be arbitrary, but the result must add up to the specified length exactly, at which point frame will be processed. Write to `tx_ctl` clears out any half-written frame that might be buffered, so writer needs to ensure there's only one frame in flight at any time.
- `5f6d4f53-5f52-5043-5f72-785f63746c5f (_mOS_RPC_rx_ctl_)` - a read/notify attribute. It returns the length of the frame that device wishes to transmit as a big-endian 32-bit number. If this value is not zero, frame data will be returned in response to read requests of the data attribute. Read chunks will be up to MTU bytes in size. Client may subscribe to notifications on this attribute. Notification will be sent whenever a new frame is submitted for delivery and the notification body will contain length (the same value as returned by reading). Upon receiving notification client can proceed to read the data without reading `rx_ctl` again.

Lightning How To

- Add - origin: <https://github.com/mongoose-os-libs/rpc-gatts> to your mos.yml libs
- For this example we'll be doing a wifi.scan as that's very handy! Add - origin: <https://github.com/mongoose-os-libs/rpc-service-wifi> to your mos.yml libs as well.
- Ensure your mos.yml config.schema has:

```
- ["bt.enable", "b", true, {title: "Enable the BT service"}]
- ["bt.keep_enabled", "b", true, {title: "Keep the BT service enabled despite WIFI being up"}]
- ["bt.random_address", "b", false, {title: "Use a consistent BT MAC address"}]
- ["bt.dev_name", "MyDeviceName_?????"]
```

- Rebuild/flash your app
- Run a BLE scanner (ie Bluesee) to see your `MyDeviceName_?????` device. Connect to see the services. Look for the one with `5f52-5043` in the middle.
- Select that service to open it. In Bluesee you'll have 3 characteristics per above docu.
- Figure out your RPC command, we'll use: `{"id":1999,"method":"Wifi.Scan"}`
- We need to tell the service ahead of time how many hex bytes we'll be sending. We need to convert ascii to hex, trim the extra spaces and then count the bytes. Fun!
- Enter your RPC command into this ascii to hex converter: <https://www.rapidtables.com/convert/number/ascii-to-hex.html>
- It'll give you a space separated hex output: `7b 22 69 64 22 3a 31 39 39 39 2c 22 6d 65 74 68 6f 64 22 3a 22 57 69 66 69 2e 53 63 6 1 6e 22 7d`
- Trim the spaces with <http://www.unit-conversion.info/texttools/replace-text/>
- Count the new hex string with: <https://mothereff.in/byte-counter>
- Halve the string count, as two hex characters equal one byte of data. With the text above, $64 / 2 = 32$.
- Convert 32 to the large 32 bit number the system is expecting. Note: It's little endian, not big endian. Use this converter: <https://www.scadacore.com/tools/programming-calculators/online-hex-converter/>. Put the 32 in the input, then copy/paste the 'UINT32 - Little Endian (DCBA)' result and trim spaces:`00000020`
- Enter that `00000020` into the write only characteristic that has `5043-5F74` in the middle. Press write. In the console you'll see a log entry like: `mgos_rpc_channel_ga:296 0x3ffe1d24 expected_flen 32 res 0`
- On the middle read only characteristic with `5043-5F72` in the middle, click subscribe. The number will change to let us know when

data is available (and how long the hex string is)

- On the read/write characteristic with 5043-5F53 in the middle, toggle the hex button so it says Ascii
- Paste in our ascii command: {"id":1999,"method":"Wifi.Scan"}
- You'll see the middle read only characteristic will have changed to let us know data is ready
- Click 'read' on the top read/write characteristic we just entered the scan command into. It'll start pulling the data of the scan results! You can click it multiple times to scroll through the data till the end.
- You can repeat the request by clicking the write only characteristic to send it our command length, then paste our command again, and click write.

Arduino SPI library for Mongoose OS

This library provides an Arduino compatibility layer for SPI by providing an SPI.h public header, so that one could pick an existing Arduino program which uses SPI, throw it into the Mongoose OS application sources, and ideally, it "just works".

AWS IoT support for Mongoose OS

This library implements integration of Mongoose OS with AWS IoT.

See tutorial at <https://mongoose-os.com/docs/mongoose-os/cloud/aws.md>

Mongoose Embedded Web Server Library

See [here](#).

BME280 sensor API

Introduction

This package contains the Bosch Sensortec's BME280 pressure sensor driver (sensor API)

The sensor driver package includes bme280.c, bme280.h and bme280_defs.h files.

Version

File	Version	Date
bme280.c	3.3.4	14 Feb 2018
bme280.h	3.3.4	14 Feb 2018
bme280_defs.h	3.3.4	14 Feb 2018

Integration details

- Integrate bme280.h, bme280_defs.h and bme280.c file in to the project.
- Include the bme280.h file in your code like below.

```
# include "bme280.h"
<p class="mume-header" id="include-bme280h"></p>
```

File information

- bme280_defs.h : This header file has the constants, macros and datatype declarations.
- bme280.h : This header file contains the declarations of the sensor driver APIs.
- bme280.c : This source file contains the definitions of the sensor driver APIs.

Supported sensor interfaces

- SPI 4-wire
- I2C

SPI 3-wire is currently not supported in the API.

Usage guide

Initializing the sensor

To initialize the sensor, user need to create a device structure. User can do this by creating an instance of the structure bme280_dev. After creating the device strcutre, user need to fill in the various parameters as shown below.

Example for SPI 4-Wire

```
struct bme280_dev dev;
int8_t rslt = BME280_OK;

/* Sensor_0 interface over SPI with native chip select line */
dev.dev_id = 0;
dev.intf = BME280_SPI_INTF;
dev.read = user_spi_read;
dev.write = user_spi_write;
dev.delay_ms = user_delay_ms;

rslt = bme280_init(&dev);
```

Example for I2C

```
struct bme280_dev dev;
int8_t rslt = BME280_OK;

dev.dev_id = BME280_I2C_ADDR_PRIM;
dev.intf = BME280_I2C_INTF;
dev.read = user_i2c_read;
dev.write = user_i2c_write;
dev.delay_ms = user_delay_ms;

rslt = bme280_init(&dev);
```

Regarding compensation functions for temperature,pressure and humidity we have two implementations.

1. Double precision floating point version
2. Integer version

By default, integer version is used in the API. If the user needs the floating point version, the user has to uncomment BME280_FLOAT_ENABLE macro in bme280_defs.h file or add that to the compiler flags.

In integer compensation functions, we also have below two implementations for pressure.

1. For 32 bit machine.
2. For 64 bit machine.

By default, 64 bit variant is used in the API. If the user wants 32 bit variant, the user can disable the macro BME280_64BIT_ENABLE in bme280_defs.h file.

Sensor data units

The sensor data units depends on the following macros being enabled or not,
(in bme280_defs.h file or as compiler macros)

- BME280_FLOAT_ENABLE
- BME280_64BIT_ENABLE

In case of the macro "BME280_FLOAT_ENABLE" enabled,

The outputs are in double and the units are

- °C for temperature
- % relative humidity
- Pascal for pressure

In case if "BME280_FLOAT_ENABLE" is not enabled, then it is

- int32_t for temperature with the units 100 * °C
- uint32_t for humidity with the units 1024 * % relative humidity
- uint32_t for pressure
 - If macro "BME280_64BIT_ENABLE" is enabled, which it is by default, the unit is 100 * Pascal
 - If this macro is disabled, Then the unit is in Pascal

Stream sensor data

Stream sensor data in forced mode

```

int8_t stream_sensor_data_forced_mode(struct bme280_dev *dev)
{
    int8_t rslt;
    uint8_t settings_sel;
    struct bme280_data comp_data;

    /* Recommended mode of operation: Indoor navigation */
    dev->settings.osr_h = BME280_OVERSAMPLING_1X;
    dev->settings.osr_p = BME280_OVERSAMPLING_16X;
    dev->settings.osr_t = BME280_OVERSAMPLING_2X;
    dev->settings.filter = BME280_FILTER_COEFF_16;

    settings_sel = BME280_OSR_PRESS_SEL | BME280_OSR_TEMP_SEL | BME280_OSR_HUM_SEL | BME280_FILTER_SEL;

    rslt = bme280_set_sensor_settings(settings_sel, dev);

    printf("Temperature, Pressure, Humidity\r\n");
    /* Continuously stream sensor data */
    while (1) {
        rslt = bme280_set_sensor_mode(BME280_FORCED_MODE, dev);
        /* Wait for the measurement to complete and print data @25Hz */
        dev->delay_ms(40);
        rslt = bme280_get_sensor_data(BME280_ALL, &comp_data, dev);
        print_sensor_data(&comp_data);
    }
    return rslt;
}

void print_sensor_data(struct bme280_data *comp_data)
{
# ifdef BME280_FLOAT_ENABLE
<p class="mume-header " id="ifdef-bme280_float_enable"></p>

    printf("%0.2f, %0.2f, %0.2f\r\n",comp_data->temperature, comp_data->pressure, comp_data->humidity);
# else
<p class="mume-header " id="else-2"></p>

    printf("%ld, %ld, %ld\r\n",comp_data->temperature, comp_data->pressure, comp_data->humidity);
# endif
<p class="mume-header " id="endif-4"></p>
}

```

Stream sensor data in normal mode

```

int8_t stream_sensor_data_normal_mode(struct bme280_dev *dev)
{
    int8_t rslt;
    uint8_t settings_sel;
    struct bme280_data comp_data;

    /* Recommended mode of operation: Indoor navigation */
    dev->settings.osr_h = BME280_OVERSAMPLING_1X;
    dev->settings.osr_p = BME280_OVERSAMPLING_16X;
    dev->settings.osr_t = BME280_OVERSAMPLING_2X;
    dev->settings.filter = BME280_FILTER_COEFF_16;
    dev->settings.standby_time = BME280_STANDBY_TIME_62_5_MS;

    settings_sel = BME280_OSР_PRESSEL;
    settings_sel |= BME280_OSР_TEMPSEL;
    settings_sel |= BME280_OSР_HUMSEL;
    settings_sel |= BME280_STANDBYSEL;
    settings_sel |= BME280_FILTERSEL;
    rslt = bme280_set_sensor_settings(settings_sel, dev);
    rslt = bme280_set_sensor_mode(BME280_NORMAL_MODE, dev);

    printf("Temperature, Pressure, Humidity\r\n");
    while (1) {
        /* Delay while the sensor completes a measurement */
        dev->delay_ms(70);
        rslt = bme280_get_sensor_data(BME280_ALL, &comp_data, dev);
        print_sensor_data(&comp_data);
    }

    return rslt;
}

void print_sensor_data(struct bme280_data *comp_data)
{
# ifdef BME280_FLOAT_ENABLE
<p class="mume-header" id="ifdef-bme280_float_enable-1"></p>

    printf("%0.2f, %0.2f, %0.2f\r\n", comp_data->temperature, comp_data->pressure, comp_data->humidity);
# else
<p class="mume-header" id="else-3"></p>

    printf("%ld, %ld, %ld\r\n", comp_data->temperature, comp_data->pressure, comp_data->humidity);
# endif
<p class="mume-header" id="endif-5"></p>
}

```

Templates for function pointers

```

void user_delay_ms(uint32_t period)
{
    /*
     * Return control or wait,
     * for a period amount of milliseconds
     */
}

int8_t user_spi_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     * to be set low to activate the relevant device on the SPI bus
     */

    /*
     * Data on the bus should be like
     * |-----+-----+-----|

```

```

* | MOSI           | MISO          | Chip Select |
* |-----+-----+-----|
* | (don't care) | (don't care) | HIGH          |
* | (reg_addr)   | (don't care) | LOW           |
* | (don't care) | (reg_data[0]) | LOW           |
* | (....)       | (....)        | LOW           |
* | (don't care) | (reg_data[len - 1]) | LOW          |
* | (don't care) | (don't care) | HIGH          |
* |-----+-----+-----|
*/
return rslt;
}

int8_t user_spi_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     * to be set low to activate the relevant device on the SPI bus
     */
    /*
     * Data on the bus should be like
     * |-----+-----+-----|
     * | MOSI       | MISO       | Chip Select |
     * |-----+-----+-----|
     * | (don't care) | (don't care) | HIGH          |
     * | (reg_addr)   | (don't care) | LOW           |
     * | (reg_data[0]) | (don't care) | LOW           |
     * | (....)       | (....)        | LOW           |
     * | (reg_data[len - 1]) | (don't care) | LOW          |
     * | (don't care) | (don't care) | HIGH          |
     * |-----+-----+-----|
     */
    return rslt;
}

int8_t user_i2c_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to store the I2C address of the device
     */
    /*
     * Data on the bus should be like
     * |-----+-----|
     * | I2C action | Data |
     * |-----+-----|
     * | Start      | -   |
     * | Write      | (reg_addr) |
     * | Stop       | -   |
     * | Start      | -   |
     * | Read       | (reg_data[0]) |
     * | Read       | (....)        |
     * | Read       | (reg_data[len - 1]) |
     * | Stop       | -   |
     * |-----+-----|
     */
    return rslt;
}

int8_t user_i2c_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*

```

```

/* The parameter dev_id can be used as a variable to store the I2C address of the device
*/
/*
 * Data on the bus should be like
 * |-----+-----|
 * | I2C action | Data |
 * |-----+-----|
 * | Start      | -   |
 * | Write       | (reg_addr) |
 * | Write       | (reg_data[0]) |
 * | Write       | (...) |
 * | Write       | (reg_data[len - 1]) |
 * | Stop        | -   |
 * |-----+-----|
 */
return rslt;
}

```

Copyright (C) 2016 - 2017 Bosch Sensortec GmbH# Bosch Sensortec's BMP280/BME280 Mongoose OS library

It uses Bosch Sensortec's reference implementation.

Both BMP280 and BME280 are supported with I2C or 4-wire SPI interface.

The library uses global instances of I2C or SPI with `spi.cs0_gpio`.

If different pins than the default ones are used, the user should define them in the `config_schema` of `mos.yml`

LwIP TCP/IP stack

[LwIP](#) version 2.1.2 + FreeRTOS bindings.

This library is used by ports other than ESP8266 and ESP32 (which ship their own, highly modified versions).

Arduino Adafruit - 16-Channel 12-bit PWM/Servo Driver - I2C interface - PCA9685 library for Mongoose OS

Tested and works on esp8266/esp32

Description

16-Channel 12-bit PWM/Servo Driver – I2C interface – PCA9685 Module lets you add more IOs to your microcontroller boards. The Module has chainable I2C interface meaning for even more IOs more boards can be daisy chained. Its an excellent product for Robots that require lots of Servo motors to control using single microcontroller. See [product page link](#) for more information about the hardware.

Features

- It is 5V compliant, which means you can control it from a 3.3V microcontroller and still safely drive up to 6V outputs (this is good for

when you want to control white or blue LEDs with 3.4+ forward voltages)

- 6 address select pins so you can wire up to 62 of these on a single i2c bus, a total of 992 outputs – that's a lot of servos or LEDs
- Adjustable frequency PWM up to about 1.6 KHz
- 12-bit resolution for each output – for servos, that means about 4us resolution at 60Hz update rate
- Configurable push-pull or open-drain output
- Output enable pin to quickly disable all the outputs
- Reverse polarity protection on the terminal block input
- Power-good LED
- 3 pin connectors in groups of 4 so you can plug in 16 servos at once
- A big capacitor on the V+ line
- 220 ohm series resistors on all the output lines to protect them, and to make driving LEDs trivial
- IO expander chip: PCA9685PW
- Operating Voltage: 2.3V – 5.5V
- Solder jumpers for the 6 address select pins
- I2C 7-bit address Range: between 0x40-0x7F, selectable with jumpers

Usage

mos.yml, add:

```
config_schema:  
- ["i2c.enable", true]  
libs:  
- origin: https://github.com/mongoose-os-libs/arduino-adafruit-pwm-servo
```

init.js, add:

```
load('api_pwm_servo.js');
```

main.c, add:

```
#include "mgos_arduino_PWMservoDriver.h"
```

Slave address

- Board 0: Address = 0x40 Offset = binary 00000 (no jumpers required)
- Board 1: Address = 0x41 Offset = binary 00001 (bridge A0)
- Board 2: Address = 0x42 Offset = binary 00010 (bridge A1)
- Board 3: Address = 0x43 Offset = binary 00011 (bridge A0 & A1)
- ...

Crontab RPC Service

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives

Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

PCF8574/PCF8575 I2C Driver

A Mongoose library for PCF857X, a popular and cheap set of GPIO extenders using I2C. The PCF8574 is an 8-port device, and the PCF8575 is a 16-port device, but they are otherwise identical.

Implementation details

The PCF857X is a pseudo bi-directional GPIO, which means that by default ports are input and output. Writes to the ports can change the state from high to low. If the port is high, it is pulled up by a weak current. Each port can be pulled low by an external device, in which case reads from that port return 0. The implication is that "input" ports are always pulled high.

API Description

To start, `mgos_pcf857x_create()` is called with the correct I2C bus and address (by default 0x20), and optionally a GPIO pin on the microcontroller that serves as an interrupt pin, to detect PCF857X input state changes.

NOTE: When the driver starts, it polls the current state from the chip without changing any ports. The benefit of this is that the MCU can safely reboot without loss of the GPIO state in PCF857X.

The API follows `mgos_gpio.h` closely, enabling ports to be set as input (ignoring the `pull_type`, which is always pullup), or as output. Notably, `mgos_pcf857x_gpio_set_int_handler()` and `mgos_pcf857x_gpio_set_button_handler()` work identically to the `mgos_gpio_*()` variants.

Example application

```
#include "mgos.h"
#include "mgos_config.h"
#include "mgos_pcf857x.h"

static void button_cb(int pin, void *user_data) {
    struct mgos_pcf857x *d = (struct mgos_pcf857x *)user_data;
    LOG(LL_INFO, ("GPIO[%d] callback, value=%d", pin, mgos_pcf857x_gpio_read(d, pin)));
    mgos_pcf857x_gpio_toggle(d, pin+4);
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_pcf857x *d;
    int i;

    if (!(d = mgos_pcf8574_create(mgos_i2c_get_global(), mgos_sys_config_get_pcf857x_i2caddr(),
        mgos_sys_config_get_pcf857x_int_gpio())) {
        LOG(LL_ERROR, ("Could not create PCF857X"));
        return MGOS_APP_INIT_ERROR;
    }

    for(i=0; i<4; i++) mgos_pcf857x_gpio_setup_input(d, i, MGOS_GPIO_PULL_UP);
    for(i=4; i<8; i++) mgos_pcf857x_gpio_set_mode(d, i, MGOS_GPIO_MODE_OUTPUT);

    mgos_pcf857x_gpio_set_button_handler(d, 0, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_NEG, 10, button_cb, d);
    mgos_pcf857x_gpio_set_button_handler(d, 1, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_POS, 10, button_cb, d);
    mgos_pcf857x_gpio_set_button_handler(d, 2, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_ANY, 10, button_cb, d);
    mgos_pcf857x_gpio_set_button_handler(d, 3, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_ANY, 10, button_cb, d);

    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose. This library adds I2C master support for the Mongoose

OS. See default pin mapping in the yml file for your platform: `mos_<PLATFORM>.yml`.

Arduino Adafruit ADS1015 12-BiT/ADS1115 16-Bit ADC - 4 Channel with Programmable Gain Amplifier library for Mongoose OS

Tested and works on esp8266/esp32

Description

Analog-to-digital converter or higher-precision ADC.

	precision	samples\sec
ADS1015	12-bit	3300
ADS1115	16-bit	860

The chip can be configured as 4 single-ended input channels, or two differential channels. As a nice bonus, it even includes a programmable gain amplifier, up to x16, to help boost up smaller single/differential signals to the full range. See [ADS1015 ADS1115](#) for more information about the hardware.

Features

- Wide supply range: 2.0V to 5.5V
- Low current consumption: Continuous Mode: Only 150uA Single-Shot Mode: Auto Shut-Down
- Programmable data rate: 8SPS to 860SPS
- Internal Low-drift voltage reference
- Internal oscillator
- Internal PGA
- I2C interface: Pin-Selectable Addresses
- Four Single-ended or two differential inputs
- Programmable comparator
- This board/chip uses I2C 7-bit addresses between 0x48-0x4B, selectable with jumpers

Usage

mos.yml, add:

```
config_schema:  
- ["i2c.enable", true]  
libs:  
- origin: https://github.com/mongoose-os-libs/arduino-adafruit-ads1x15
```

init.js, add:

```
load('api_ads1015.js');
```

[\(JavaScript\) usage example](#)

main.c, add:

```
# include "mgos_arduino_Adafruit_AMG8833.h"
<p class="mume-header" id="include-mgos_arduino_amg8833h"></p>
```

Slave address

- Address pin connected to GND = 0x48 Offset = binary 1001000
- Address pin connected to VDD = 0x49 Offset = binary 1001001
- Address pin connected to SDA = 0x4A Offset = binary 1001010
- Address pin connected to SCL = 0x4B Offset = binary 1001011

RPC Service - GPIO

This service provides an ability to manage GPIO on devices remotely.

It is possible to call this service programmatically via serial, HTTP/RESTful, WebSocket, MQTT or other transports
(see [RPC section](#)) or use `mos` tool.

Below is a list of exported RPC methods and arguments:

GPIO.Read

Set given pin in INPUT mode, read GPIO pin, return its value. Arguments:

```
{
  "pin": 15      // Required. Pin number.
}
```

Example usage:

```
mos call GPIO.Read '{"pin": 0}'
{
  "value": 1
}
```

GPIO.Write

Set given pin in OUTPUT mode, set GPIO pin. Arguments:

```
{
  "pin": 15,    // Required. Pin number.
  "value": 0    // Required. Voltage level. Either 0 (low) or 1 (high).
}
```

Example usage:

```
mos call GPIO.Write '{"pin": 2, "value": 0}'
```

GPIO.Toggle

Set given pin in OUTPUT mode, toggle voltage level and return that level. Arguments:

```
{  
    "pin": 15      // Required. Pin number.  
}
```

Example usage:

```
mos call GPIO.Toggle '{"pin": 2}'  
{  
    "value": 1  
}
```

Mongoose OS documentation

Mongoose OS is a firmware development framework for microcontrollers.

Mongoose OS was created for developers who work on commercial connected products, and care about things like manageability, security, and reliability. 80-90% of the commercial firmware is around infrastructure, which is the same for many products regardless of what they do. Mongoose OS provides such a reliable, field-tested infrastructure.

Quick Summary

- Designed for commercial products
- Reliable OTA
- Support for major cloud services: AWS IoT, Microsoft Azure, Google IoT Core, IBM Watson, Samsung Artik
- Support for generic in-house MQTT, RESTful, Websocket servers
- Networking core is based on the mature [Mongoose Networking Library](#)
- RPC infrastructure for remote management:
 - JSON-RPC 2.0 framing
 - transports: MQTT, UART, BLE, REST, Websocket
 - many built-in services like Config, FS, GPIO, I2C, GATTC, Wifi, etc
- Configuration infrastructure with "reset to factory defaults"
- Advanced security features
 - ECC508A crypto element support
 - TLS 1.2 based on the ARM mbedTLS
 - low footprint tuning of the TLS stack
- Modular: [small core](#) and [over a hundred libraries](#)
- Commercial friendly Apache 2.0 license

Smart Light IoT product reference

We have published an end-to-end IoT product reference, <https://github.com/cesanta/mongoose-os-smart-light>. It could be used as a baseline for your own commercial product! See usage guide video:

Other resources

- [Developer forum](#)
- [Developer chat](#)
- [Youtube video tutorials](#)

This documentation is browseable online at <https://mongoose-os.com/docs>

UART gateway

Introduction

UART gateway connects any device to the Internet via UART.

Everything that your device sends to the UART, the gateway forwards to the Internet server - for example, AWS IoT, or Azure, or your own private TCP / Websocket backend.

Likewise, everything that is sent to the gateway, gateway forwards to the UART.

The UART gateway could report to the BLE central, to the raw TCP server, to the Websocket server, and to the MQTT server. Also, it can accept an incoming TCP/Websocket connections. Thus it is very versatile in terms of the connection options.

UART gateway does not implement any logic whilst forwarding data to and from the UART. Data is transferred as-is, no modification happens in any direction. Message boundaries are determined by timing.

Authentication and encryption is provided by the TLS layer.

The server side must be a TLS1.2 server with client-side authentication (mutual TLS) for the industry-standard level of security. AWS IoT is an example of such service.

Quick start

1. Buy hardware mentioned on the [Uart gateway page](#)
2. Connect hardware to your workstation via USB
3. Follow steps 1,2,3 of [mos setup](#)
4. In the `mos` UI, run `mos flash https://mongoose-os.com/downloads/uart-gateway/uart-gateway.zip` command
5. Follow step 7 of [mos setup](#)

6. Login to the [license manager](#) and buy uart-gateway licenses (one per device)
7. Execute `mos license`
8. Configure MQTT:
 - For generic/private MQTT server, run

```
mos config-set mqtt.enable=true mqtt.server=HOST:PORT tu.mqtt.enable=true
```

 - For AWS, follow [AWS guide's](#) "Setup AWS IoT" and "Setup device" chapters
 - For Google, follow [GCP guide's](#) "Setup Google IoT Core" and "Setup device" chapters
 - For Azure, follow [Azure guide's](#) "Setup Azure IoT Hub" and "Setup device" chapters
 - For Watson, follow [Watson guide's](#) "Quick setup" chapter
 - For mDash, follow step 8 of [mos setup](#)
9. Connect UART: pin 25 to TX, pin 26 to RX

When done, you should have your device flashed, provisioned to WiFi, connected to the cloud, and reporting UART data. Below is an example of the eval ESP32 Devkit-C board that connects a USB-to-Serial, making computer's serial device controlled via MQTT:

USB-to-Serial converter	UART gateway on ESP32 Devkit-C
GND	GND
RX	IO25
TX	IO26

reporting data to the <http://www.mqtt-dashboard.com>:

Configuring UART gateway

UART gateway keeps its configuration on a flash filesystem. It could be instected and changed using `mos` tool. To see an existing configuration, execute `mos config-get`. Below is the documentation for relevant entries:

```

"tu": {
  "bt": { // BLE settings
    "rx_buf_size": 2048, // BLE RX buffer size
    "tx_buf_size": 2048 // BLE TX buffer size
  },
  "evict_old": true,
  "mqtt": { // MQTT settings
    "enable": true, // Enable MQTT
    "qos": 0, // MQTT publishing QoS
    "rx_topic": "uart-bridge/rx", // RX topic
    "tx_topic": "uart-bridge/tx" // TX topic
  }
},
"tcp": { // TCP/Websocket settings
  "client": { // Client mode
    "reconnect_interval": 5, // Reconnection interval, seconds
    "remote_addr": "mongoose-os.com:443", // Server HOST:PORT
    "tls": { // TLS settings
      "ca_cert": "ca.pem",
      "cert": "",
      "server_name": "mongoose-os.com"
    },
    "ws": { // Enable websocket mode
      "enable": true,
      "protocol": "",
      "uri": "/uart-to-websocket/device/?????????????" // URI to connect to
    }
  },
  "evict_old": true, // On device ID conflict, evict old entry
  "keepalive": { // Heartbeat settings
    "count": 10,
    "idle": 60,
    "interval": 6
  },
  "listener": { // Server mode settings
    "port": 8910, // Port to listen on
    "tls": { // TLS settings
      "ca_cert": "", // CA certificate for mutual auth
      "cert": "" // Server certificate
    },
    "ws": { // Enable websocket mode
      "enable": false
    }
  },
  "rx_buf_size": 2048, // Receive buffer size, bytes
  "status_interval_ms": 1000, // Status interval, milliseconds
  "tx_buf_size": 2048 // Transmit buffer size, bytes
},
"uart": { // UART settings
  "baud_rate": 115200, // Baud rate
  "rx_buf_size": 1460, // Receive buffer size, bytes
  "rx_fc_ena": false, // Enable flow control
  "rx_linger_micros": 20, // Microseconds to continue reading before sending
  "rx_throttle_when_no_net": true, // Throttle when no network
  "status_interval_ms": 1000, // Status interval, milliseconds
  "swap_rxcts_txrts": false, // Swap RX / CTS and TX / RTS
  "tx_buf_size": 2048, // Transmit buffer size
  "tx_fc_ena": false, // Enable TX flow control
  "uart_no": -1 // UART number, -1 to disable
}

```

In order to change any configuration parameter, execute `mos config-set name=value`, for example:

```

mos config-set mqtt.server=my.server.com:1883
mos config-set kiwi.pub_topic=my_cool_topic

```

Web UI

UART gateway, when not configured, starts its own WiFi Access Point, named `uart-gateway-.....`. The gateway is accessible on IP address 192.168.4.1 in that network.

If WiFi station is configured, then the UART gateway does not start an access point.

In any case, UART gateway runs a Web UI on port 80, allowing to setup tunable parameters over the easy-to-use interface:

- [User guide](intro.md) # Libraries

Mongoose OS libraries are maintained as separate repositories under the [mongoose-os-libs](#) organization.

Here's a list as of 2019/04/30:

Name	Description
adc	ADC support
ads1x1x-i2c	ADS101X and ADS111X I2C drivers
ads7843-spi	
apds9960-i2c	Mongoose OS library for APDS-9960
arduino-adafruit-ads1x15	Arduino Adafruit ADS1015 library for Mongoose OS
arduino-adafruit-bme280	
arduino-adafruit-gfx	
arduino-adafruit-htu21df	
arduino-adafruit-pwm-servo	Arduino Adafruit PWM servo library
arduino-adafruit(ssd1306)	
arduino-adafruit-tsl2561	
arduino-compat	Arduino compatibility library for Mongoose OS
arduino-dallas-temperature	
arduino-ds3231	Arduino DS3231 library for Mongoose OS
arduino-md-parola	
arduino-onewire	
arduino-pololu-vl53l0x	
arduino-spi	
arduino-wire	
atca	
aws	Amazon AWS IoT integration with Mongoose OS
azure	Azure IoT Hub support library for Mongoose OS
barometer	Mongoose OS API and library for barometers
bh1730	BH1730 sensor support
bh1750	
blynk	Blynk integration library for Mongoose OS

bm222	A Bosch BM222 accelerometer library
bme280	
boards	Board configuration defs
bootloader	Boot loader support library
bt-common	Common bits of Bluetooth support
bt-service-config	System Configuration over Bluetooth GATT Service
bt-service-debug	Debug over Bluetooth GATT Service
bt-service-time	Bluetooth current time service implementation
ca-bundle	
ccs811-i2c	Mongoose OS library implementing an I2C driver for CCS811
core	Mongoose OS core lib
cron	
crontab	
dallas-rmt	
dash	
demo-bundle	
dht	
dns-sd	
ds3231	
empty	A blank Mongoose OS library
esp32-touchpad	JS bindings for ESP32 touch pad sensor
ethernet	Ethernet support
file-logger	
freertos	FreeRTOS Kernel
fstab	Storage device and filesystem init table
gcp	Google IoT Core integration for Mongoose OS
http-server	
htu21df-i2c	Mongoose OS library implementing an I2C driver for HTU21D(F)
i2c	
ili9341-spi	ILI9341 TFT Driver for Mongoose OS
imu	Mongoose OS library for inertial measurement units
ina219-i2c	
ina226-i2c	I2C Library for Texas Instruments' current sensor INA226
ina3221-i2c	I2C Library for Texas Instruments' 3-channel current sensor INA3221
ir	IR library for Mongoose OS
js-demo-bundle	

jstore	
location	
lwip	LwIP TCP/IP stack
max7219-spi	MAX7219/MAX7221 SPI based LED display driver.
mbedtls	Optimized version of mbedtls
mcp23xxx	Mongoose OS Library for MCP23x08/MCP23x17: an 8/16-port GPIO expander w/ interrupt support
mcp9808-i2c	Mongoose OS library implementing an I2C driver for MCP9808
md-max72xx	
miniz	Miniz compression library
mjs	Embedded JavaScript engine for Mongoose OS
mongoose	Mongoose Embedded Web Server Library
mq135	MQ135 gas sensor support
mqtt	MQTT protocol support library for Mongoose OS
neopixel	
onewire	
onewire-rmt	
ota-common	OTA common bits
ota-http-client	
ota-http-server	
ota-shadow	
pcf857x-i2c	Mongoose OS Library for PCF8574: an 8-port GPIO expander w/ interrupt support
pppos	PPPoS / cellular modem support
prometheus-metrics	
prometheus-sensors	A library that takes measurements of popular sensors on I2C/SPI and forwards them to Prometheus
provision	A device provisioning library
pwm	
rpc-azure	RPC support for Azure
rpc-common	
rpc-gatts	RPC over GATT, Server
rpc-gcp	RPC support for Google Cloud Platform
rpc-loopback	
rpc-mqtt	Implementation of Mongoose OS RPC over MQTT protocol
rpc-service-atca	
rpc-service-config	
rpc-service-cron	
rpc-service-fs	

rpc-service-gattc	
rpc-service-gpio	
rpc-service-i2c	
rpc-service-ota	
rpc-service-wifi	
rpc-uart	
rpc-ws	WebSocket channel support for mgRPC
sensor-utils	For Mongoose OS - A collection of C-functions with JS wrappers that make it convenient to calculate ...
shadow	
sht31-i2c	Mongoose OS library implementing an I2C driver for SHT31
si7005	Si7005 Temperature and Humidity Sensor Driver for Mongoose OS
si7021-i2c	Mongoose OS library implementing an I2C driver for Si7021
sntp	
spi	Implements SPI API for Mongoose OS
ssd1306	
stlm75	STMicro STLM75 temperature sensor driver
stmpe610-spi	
tmp006	A Texas Instruments TMP006 temperature sensor
veml6075-i2c	
vfs-common	Mongoose OS VFS common bits
vfs-dev-encr	VFS device that encrypts reads and writes
vfs-dev-part	A VFS device that exposes a part of another device
vfs-dev-ram	RAM-based VFS device
vfs-dev-spi-flash	SPI flash VFS device driver
vfs-dev-w25xxx	Winbond W25XXX SPI NAND Flash Driver
vfs-fs-lfs	Mongoose OS support for littlefs
vfs-fs-spiffs	SPIFFS filesystem support
watson	IBM Watson IoT Platform support for Mongoose OS
wifi	
wifi-ism43xxx	Inventek es-WiFi module driver for Mongoose OS
wifi-setup-web-ui	

Heaplog viewer quick overview

So typical use case for the heaplog viewer is when we get some heap integrity violations, something like this in the log:

```
heap integrity broken: block links don't match: 1110 -> 1111, but 1111 -> 0
```

So we enable poisoning, heaplog, and call stack trace, in order to understand better what's going on:

```
$ mos build --build-var MGOS_ENABLE_HEAP_LOG:1 --build-var MGOS_ENABLE_CALL_TRACE:1
```

Before flashing this firmware, make sure you'll be able to save full session's log into a file. I usually adjust MFT `console-line-count` to be really large, say, 50000 lines, and clear the console before flashing the device. It will be easy to just copy-paste the whole log afterwards. Alternatively, you may use `console-log` option of MFT to save the log to the file for you, although you'll have to clean it yourself before each new session.

Then, flash device and reproduce the problem. Now, instead of "heap integrity broken" message, we should get something more concrete about the poison being missing at some exact address, for example:

```
there is no poison after the block. Expected poison address: 0x3fff3dfc,  
actual data: 0x1f 0x00
```

Once reproduced, copy-paste complete session's log from MFT to some file. It should typically begin with something like:

```
--- flashed successfully  
--- connected  
[some garbage omitted]  
rl!◆hlog_param:{"heap_start":0x3fff01b0, "heap_end":0x3fffc000}
```

The key thing here is `hlog_param:{"heap_start":0x3fff01b0, "heap_end":0x3fffc000}`

Now, it's very useful to convert all FW addresses to symbolic names. There is a script `heaplog_symbolize.py` for that (located in the same directory as this readme file: `tools/heaplog_viewer`).

Using the heap log server

Heap log server, located under the `heaplog_server` directory, simplifies the process of collecting logs and providing symbols. Once built (you'll need to install [Go](#), you can run it like this:

```
./heaplog_server --logtostderr \  
--document_root .. \  
--binary $(HOME)/cesanta/dev/fw/platforms/esp8266/.build/mongoose-os.out \  
--console_log /tmp/console.log
```

Point it at the firmware binary and console log. Console log can be a serial port or a file that you collected before. After that, you can navigate to <http://localhost:8910/> and use the "Connect to server" button, viewer will load the symbols and log automatically.

If the log file is a file (and not a port), the server tries to automatically find the latest heap log start and uses it as a starting point, so you do not need to clean up the log every time.

Symbolizing manually

Assuming you saved a heaplog to `/tmp/log1`, a command to symbolize it would be:

```
python heaplog_symbolize.py --out-suffix _symb /tmp/log1
```

It will create a new file `/tmp/log1_symb`. If you omit `--out-suffix` option, output will go to stdout.

So now you have a heaplog with symbolic call traces. Open heaplog viewer in your browser: `tools/heaplog_viewer/heaplog_viewer.html` (If you had it already opened with some other data, please refresh it by hitting F5 before loading new file, since viewer is not yet polished)

Click "Choose File", and select the file with log you just saved. After a few seconds, you should see a visual heap map.

NOTE: If the map doesn't appear, your log file might be corrupted, or you might forget to hit F5 before loading new heaplog file. To check if there are some errors, open developers console (in Opera or Chrome, it's Ctrl+Shift+J), and check if there are any uncaught exceptions. If help is needed, ping `dfrank` in Slack.

So now you have a visual heap map and an address where poison is missing (in the example above, it's `0x3fff3dfc`). By moving your mouse over the map, you'll see corresponding addresses. This way it's easy to find the offending address on the map; usually, it's the end of some block. Now we know that this particular block was overflowed. In the details, there will be call stack like the following:

```
# 4849 Alloc: 0x3fff70ac, size: 1024, shim: 1, calls: __wrap_pvPortMalloc ←
<p class="mume-header" id="4849-alloc-0x3fff70ac-size-1024-shim-1-calls-__wrap_pvportmalloc"></p>
malloc ← _malloc_r ← cs_read_file ← init_device
```

Having that information now, it's much easier for you to find a bug. Good luck!

Shortening the log

If the log is too long to be parsed by the heaplog viewer, there is a standalone shortener script which takes a heaplog, parses it, and swallows all the history, leaving just a set of `mallocs` which reconstruct the final picture from the log. Usage:

```
$ cd tools/heaplog_viewer/heaplog_shortener && \
go build && \
./heaplog_shortener --console_log /path/to/src_log > target_short_log
```

External dependencies for Docker image builds get put here.

Applications

Mongoose OS apps are maintained as separate repositories under the [mongoose-os-apps](#) organization.

Here's a list as of 2019/04/30:

Name	Description
LilyGo-HiGrow-Sensor-v1	LilyGo HiGrow ESP32 Plant Monitoring Sensor Firmware (Hardware v1)

alexamicrowave	Control a microwave with ESP32 via MQTT / Amazon Alexa
aws-iot-button	
aws-iot-heater	
aws-pico	ESP32-PICO-KIT breakout board demo for AWS IoT
aws-uart	
blynk	
blynk-bme280-js	Using BME280 sensor with Blynk mobile app
bootloader	Boot loader
captive-portal	
demo-c	A demo Mongoose OS firmware for C/C++
demo-js	A JS-enabled demo Mongoose OS firmware
door-bell	IoT doorbell that works via MQTT - generic MQTT server, AWS IoT, Google IoT, etc
door-sensor	Mongoose OS IoT Door-Sensor for ESP32
empty	
environmental-monitor	
example-arduino-adafruit-ads1x15-js	Adafruit ADS1015 12-Bit ADC - 4 Channel with Programmable Gain Amplifier usage example (JavaScript)
example-arduino-adafruit-bme280-c	
example-arduino-adafruit-bme280-js	
example-arduino-adafruit-ssd1306-c	
example-arduino-adafruit-ssd1306-js	
example-arduino-dallas-temperature-c	
example-arduino-dallas-temperature-js	
example-arduino-ds3231-js	DS3231 High Precision Real Time Clock Module usage example (JavaScript)
example-arduino-hello-c	
example-arduino-onewire-c	
example-arduino-onewire-js	
example-arduino-wire-c	
example-binary-lib-c	Binary library use example
example-ble-js	BLE JS API use example
example-dht-c	
example-dht-js	
example-dns-sd	DNS-SD usage example
example-m5stack-js	Basic demo for M5stack
example-neopixel-c	
example-neopixel-js	

example-no-libs-c	Bare-bones app with no libraries at all
example-rpc-c	
example-shadow-js	
example-spi-c	C SPI API usage example
example-uart-c	
example-uart-js	
example-wifi-scan-js	WiFi scan app in JS
http-fetch	
huzzah-featherwing	
imu-demo	IMU Library Demo (for Chrome)
lilygo-higrow-soil-temp-humi-sensor	
losant-motion-sensor	Detecting Motion Using a PIR sensor, ESP8266, and Mongoose OS
losant-mqtt	Losant + Mongoose OS example
losant-temp-sensor	A Mongoose OS app for sending temperature readings of ESP devices to Losant.
mq135	
neopixel-aws-iot	Neopixel control using AWS IoT and Android App
ota-aws-shadow	
shelly1	
smallest-wifi-button	The Smallest WiFi Button in the World (ESP8266, MQTT, IFTTT)
smart-pool-pump	Pool Timer Controller, integrates AWS shadow, BMP180 and ADE7912
smart-washing-machine	Making washing machine smart - MQTT, ESP8266
smoke-detector	IoT Smoke Detector Tutorial (IFTTT, SMS)
sonoff-basic-openhab	A Sonoff Basic firmware to work with openHAB
wifi-scan	
wifi-setup-web	

People who have agreed to the

[Cesanta CLA](#)

can make contributions.

We follow the Google C/C++ style guide: <https://google.github.io/styleguide/cppguide.html>

We'd appreciate if your contribution follows the same style guide.

Mongoose OS - an IoT Firmware Development Framework

- Over-The-Air firmware updates and remote management - reliable updates with rollback on failures, remote device access infrastructure
- Security - built in flash encryption, crypto chip support, ARM mbedTLS optimized for small memory footprint
- [Device management dashboard service](#)

- Supported microcontrollers: CC3220, CC3200, ESP32, ESP8266, STM32F4, STM32L4, STM32F7
- Recommended dev kits: [ESP32-DevKitC for AWS IoT](#), [ESP32 Kit for Google IoT Core](#)
- Built-in integration for AWS IoT, Google IoT Core, Microsoft Azure, Adafruit IO, generic MQTT servers
- Code in C or JavaScript
- Ready to go Apps and Libraries
- [Embedded JavaScript engine - mJS](#)

Trusted and Recommended By:

- Amazon AWS - [Amazon AWS Technology Partner](#)
- Google IoT Core - [Mongoose OS is a Google Cloud IoT Core Partner](#)
- IBM Watson IoT - [Mongoose OS is a Ready for IBM Watson IoT validated solution](#)
- Microsoft Azure IoT - [Mongoose OS is recommended by Microsoft Azure IoT](#)
- Texas Instruments - [an official partner of Texas Instruments](#)
- STMicroelectronics - [an official partner of STMicroelectronics](#)
- Espressif Systems - [an official partner of Espressif Systems](#)

Docs, Support

- [Mongoose OS Documentation](#)
- [Support Forum](#) - ask your technical questions here
- [Video tutorials](#)
- [Commercial licensing](#) and [support available](#)

Licensing

Mongoose OS is Open Source and dual-licensed:

- **Mongoose OS Community Edition** - Apache License Version 2.0
- **Mongoose OS Enterprise Edition** - Commercial License

Community vs Enterprise Edition

	Community Edition	Enterprise Edition
License	Apache 2.0	Commercial - contact us
Allows to close end-product's source code	Yes	Yes
Price	Free	Paid, see details
Source code & functionality	Limited	Full
Technical support	Community support via Forum and Chat	Commercial support by Mongoose OS development team, see details

How to contribute

- If you have not done it already, sign [Cesanta CLA](#) and send GitHub pull request.
- Make a Pull Request (PR) against this repo. Please follow [Google Coding Style](#).

Send PR to one of the core team member:

- [pimvanpelt](#)
- [nliviu](#)
- [DrBomb](#)
- [kzyapkov](#)
- [rojer](#)
- [cpq](#)

- Responsibilities of the core team members:

- Review and merge PR submissions
- Create new repos in the <https://github.com/mongoose-os-apps> and <https://github.com/mongoose-os-libs> organisations for new app/library contributions
- Create Mongoose OS releases

Flashing ESP8266

Wiring

At the bare minimum, you need to have serial TX & RX connected to the RX and TX pins on ESP8266 board. To enter flashing mode GPIO0 must be pulled down on boot, so you can simply connect it to GND before connecting power.

Other option is to connect GPIO0 to DTR (Data Terminal Ready) line of serial adapter and RESET pin to RTS (Request To Send) line. This is the way it is wired in NodeMCU boards and what `esptool.py` also assumes.

Summary:

- GND - ground
- TX - serial RX
- RX - serial TX
- VCC - power
- CH_PD - power
- GPIO0 - ground or serial DTR
- RESET - leave unconnected or serial RTS

Drivers

Don't forget to install proper drivers for the USB-to-serial converter.

If you're using a board along with a standalone FTDI adapter on Linux or Mac OS X, it should be shipped with the OS, for Windows drivers see [FTDI website](#).

NodeMCU v2 boards come with Silabs CP2102 USB-to-serial chip for which you can get driver from [here](#).

NodeMCU 0.9 and some other boards use the CH34x chip, Win7 driver can be found [here](#).

Mongoose Flashing Tool

If you have DTR and RTS connected properly, pressing "Load firmware" button should reboot the device automatically. Otherwise, just connect GPIO0 to ground and reset the device manually before flashing.

esptool

As `esptool.py` does not generate the device ID, you need to do this manually if you want to connect the device to our cloud. To do this invoke `mkid.py` script:

```
./tools/mkid.py --id ${ID} --psk ${PSK} > 0x10000.bin
```

Replace `${ID}` with a few random characters and `${PSK}` with a password unique to this device. Alternatively, you can generate a random ID with MFT:

```
MFT --generate-id 0x10000.bin
```

After that just flash `0x10000.bin` at offset `0x10000` along with other pieces:

```
path/to/esptool.py -b 115200 -p /dev/ttyUSB0 write_flash 0x00000 0x00000.bin 0x10000 0x10000.bin 0x1d000 0x1d000.bin 0x6d000 0x6d000.bin
```

For NodeMCU v2 board don't forget to also add

`--flash_mode dio --flash_size 32m` flags after `write_flash` or the device won't boot.

```
/*
```

- Copyright (c) 2016, Texas Instruments Incorporated
- All rights reserved.
-
- Redistribution and use in source and binary forms, with or without
- modification, are permitted provided that the following conditions
- are met:
-
- ◦ Redistributions of source code must retain the above copyright
- notice, this list of conditions and the following disclaimer.
-
- ◦ Redistributions in binary form must reproduce the above copyright
- notice, this list of conditions and the following disclaimer in the
- documentation and/or other materials provided with the distribution.
-
- ◦ Neither the name of Texas Instruments Incorporated nor the names of
- its contributors may be used to endorse or promote products derived
- from this software without specific prior written permission.
-
- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
- AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
- THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
- PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
- CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
- EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
- PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
- OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
- WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
- OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
- EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
- /
- /
- ===== CC3220SF_LAUNCHXL.cmd =====
- */

```
--stack_size=1024
--heap_size=0 /* minimize heap since we are using heap_4.c */
--entry_point=_c_int00

/*
• The starting address of the application. Normally the interrupt vectors
• must be located at the beginning of the application.
*/
#define SRAM_BASE 0x20000000
#define FLASH_BASE 0x01000800

MEMORY
{
/* Bootloader uses FLASH_HDR during initialization /
FLASH_HDR (RX) : origin = 0x01000000, length = 0x7FF / 2 KB /
FLASH (RX) : origin = 0x01000800, length = 0x0FF800 / 1022KB /
SRAM (RWX) : origin = 0x20000000, length = 0x00040000 / 256KB */
}

/* Section allocation in memory */

SECTIONS
{
.dbghdr : > FLASH_HDR
.binit : > FLASH
.text : > FLASH
.text.* : > FLASH
.const : > FLASH
.cinit : > FLASH
.pinit : > FLASH
.init_array : > FLASH

GROUP {
    .int_vecs
    .TI.ramfunc .iram .iram.*
} load=FLASH, run=SRAM_BASE, table(BINIT)
GROUP {
    .bss
    .data
    .heap_start
} > SRAM
GROUP {
    .heap_end
    .stack
} > SRAM(HIGH)
}

}
```

Mongoose OS Firmware boot loader for CC3200

`mg-boot.bin` should be loaded at `0x20004000` (this is where ROM loads `/sys/mcuimg.bin`).

It consists of a small relocator followed by the loader body.

The relocator, which is executed first, moves the loader body to `0x2003C000` (256K - 16K) and executes it from there.

The loader's stack starts at `0x2003C000` and grows down.

Loader then proceeds to initialize NWP and read its configuration.

Configuration, in the form of `struct boot_cfg`, is read either from `mg-boot.cfg.0` or `.1`. If both exist, whichever is more recent is used. Recency is determined by comparing the `seq` counter. NB: config with *smaller* `seq` is deemed to be more recent (counts down from `BOOT_CFG_INITIAL_SEQ`).

The config specifies which image to load next (`image_file`) and where to (`base_address`). As usual, it is assumed that the image starts with interrupt stack pointer followed by interrupt vector table.

Relocating loader to the end of RAM allows using the top 16 KB so application images starting at `0x20000000` are perfectly fine.

Images up to approximately 240KB (256KB - 16KB - loader stack) can be loaded this way.

Boot configs can be created with the `mkcfg` tool located in the `tools` dir:

```
$ mkcfg $IMAGE_FILE $BASE_ADDRESS > boot.cfg.0
```

This initializes `seq` to `BOOT_CFG_INITIAL_SEQ`. If specifying `seq` is required:

```
$ mkcfg $IMAGE_FILE $BASE_ADDRESS $SEQ > boot.cfg.0
```

CC3200 ROM communication protocol

All communication is done over one serial port at baud rate 921600 (no autodetection is performed). In most situations data is transmitted in frames, each frame must be acknowledged by the remote side (by sending 2 bytes: "\x00\xCC").

Frame format

Frames are length-prefixed, with no delimiters inbetween.

Length	Description
2 bytes	big-endian number, length of payload plus length of this field
1 byte	checksum, sum of all payload bytes modulo 256
N bytes	payload

First byte of the payload usually is an opcode, rest is the arguments (numbers are big-endian, if not stated explicitly).

Before sending any frames to a freshly booted board you need to send break first, then read back one ACK ("\x00\xCC").

Known opcodes

Opcodes are sorted by numerical value.

- `0x21` – start file upload.
 - Arguments:
 - 2 bytes: 0
 - 1 byte: `0x30` + block size (0 – 256, 1 – 1024, 2 – 4096, 3 – 16384)

- 1 byte: number of blocks (in terms of the above block size)
 - N bytes: target filename (usually `/sys/mcuimg.bin`) followed by 2 zero bytes
 - After sending ACK the device will send 4 zero bytes.
- `0x22` – finish file upload.
 - Arguments:
 - 63 bytes: all 0
 - 256 bytes: all `0x46`
 - 1 byte: 0
- `0x24` – file chunk. Each chunk carries 4096 bytes of the file (not the block size specified in `0x21`).
 - Arguments:
 - 4 bytes: offset of the chunk
 - 4096 bytes: data
- `0x2A` – get file info.
 - Arguments:
 - 4 bytes: name length
 - N bytes: file name
 - Response:
 - 24 bytes: first byte is 1 if the file exists, 4 bytes at offset 4 – file size
- `0x2B` – get file chunk.
 - Arguments:
 - 4 bytes: offset
 - 4 bytes: number of bytes to read
 - Response:
 - N bytes: content
- `0x2D` – raw storage write.
 - Arguments:
 - 4 bytes: storage ID – 0
 - 4 bytes: offset
 - 4 bytes: length
 - N bytes: data
- `0x2E` – file deletion.
 - Arguments:
 - 4 bytes: 0
 - N bytes: zero-terminated filename.
- `0x2F` – version info.
 - Response is sent in a new frame that you must ACK:
 - 28 bytes: some structure with first 4 bytes containing bootloader version.
- `0x30` – raw storage block erasing.
 - Arguments:
 - 4 bytes: storage ID – 0
 - 4 bytes: index of the first block to erase (block size can be determined with command `0x31`)
 - 4 bytes: number of blocks to erase
- `0x31` – get storage info.
 - Arguments:
 - 4 bytes: storage ID – 0
 - Response is sent in a new frame that you must ACK:
 - 2 bytes: block size
 - 2 bytes: number of blocks
- `0x32` – exec. Sometimes you need to read 2 ACKs instead of one.
- `0x33` – switch UART pins to another port.
 - Arguments:
 - 4 bytes: magic number `0x0196E6AB`

- Copyright (c) 2014-2016 Cesanta Software Limited
 - All rights reserved
 -
 - Linker command file for the TI compiler.
- */

```
--retain=g_pfnVectors

/* External vars: APP_ORG SRAM_BASE_ADDR SRAM_SIZE */

/* System memory map */

MEMORY
{
SRAM (RWX) : origin = APP_ORG, length = (SRAM_SIZE - (APP_ORG - SRAM_BASE_ADDR))
}

/* Section allocation in memory */

SECTIONS
{
.intvecs: > APP_ORG
GROUP {
.init_array
.vtable
.iram .iram.*
.text .text.*
.const
.data
.bss_start
.bss
.bss_end
.pinit
.heap_start
} > SRAM
GROUP {
.sysmem : type = DSECT
.heap_end
.stack
} > SRAM(HIGH)
}
```

Simple device control using MQTT

This example demonstrates how to perform remote device control over MQTT protocol. Build & flash a firmware and configure WiFi (see [quick start guide](#)), and configure MQTT server:

```
mos build --arch esp8266
mos flash
mos config-set wifi.....
mos console
```

Then use an MQTT client to send messages, for example web-based client at <http://www.hivemq.com/demos/websocket-client/>. Click on "Connect", then

"Add new topic subscription", choose topic `/response`, then publish a message. In the "Topic" field, specify `/request`, and in the "Message" field, one of the following:

- `{"gpio": {"pin": 2, "state": 0}}` - sets GPIO pin X into state Y
- `{"button": {"pin": 0}}` - start listening on button presses on GPIO pin X. When button is pressed, a message "click" is sent to the `/response` topic
- `{"i2c_write": {"data": "af404c"}}` - write byte stream to the I2C bus.
In this example, there are 3 bytes `0xaf 0x40 0x4c`.
First byte is an I2C address. Status code 0 in response means success, all other codes mean failure.

By default, this firmware connects to the `broker.mqttdashboard.com:1883` MQTT server.

You can change this default setting by editing `src/conf_schema.yaml` file and rebuilding/reflashing the firmware. Alternatively, you can change any setting on the fly, for example to choose a different MQTT server:

```
mos config-set mqtt.server=my_server:my_port
```

How to use Amazon IoT with this example

First, download the Amazon's `aws` utility and run `aws configure`.

Then you're ready to onboard

```
... build, flash, configure wifi  
mos aws-iot-setup --aws-iot-policy YOUR_POLICY  
mos console
```

Login to AWS IoT console, use web-based MQTT client to send messages.

See <https://mongoose-os.com/blog/esp8266-remote-control-via-aws-iot/>

for the step-by-step instructions.



JSON parser and emitter for C/C++

Features

- ISO C and ISO C++ compliant portable code
- Very small footprint
- No dependencies
- `json_scanf()` scans a string directly into C/C++ variables
- `json_printf()` prints C/C++ variables directly into an output stream
- `json_setf()` modifies an existing JSON string
- `json_fread()` reads JSON from a file
- `json_fprintf()` writes JSON to a file
- Built-in base64 encoder and decoder for binary data
- Parser provides low-level callback API and high-level `scanf`-like API
- 100% test coverage

- Used in [Mongoose OS](#), an operating system
for connected commercial products on low-power microcontrollers

API reference

json_scanf(), json_vscanf

```
int json_scanf(const char *str, int str_len, const char *fmt, ...);
int json_vscanf(const char *str, int str_len, const char *fmt, va_list ap);

/* json_scanf's %M handler */
typedef void (*json_scanner_t)(const char *str, int len, void *user_data);
```

Scans the JSON string `str`, performing scanf-like conversions according to `fmt`.

`fmt` uses `scanf()`-like format, with the following differences:

- Object keys in the format string don't have to be quoted, e.g. "{key: %d}"
- Order of keys in the format string does not matter, and the format string may omit keys to fetch only those that are of interest, for example,
assume `str` is a JSON string `{ "a": 123, "b": "hi", c: true }`.

We can fetch only the value of the `c` key:

```
int value = 0;
json_scanf(str, strlen(str), "{c: %B}", &value);
```

- Several extra format specifiers are supported:

- %B: consumes `int *` (or `char *`, if `sizeof(bool) == sizeof(char)`), expects boolean `true` or `false`.
- %Q: consumes `char **`, expects quoted, JSON-encoded string. Scanned string is malloc-ed, caller must free() the string.
- %V: consumes `char **, int *`. Expects base64-encoded string. Result string is base64-decoded, malloced and NUL-terminated. The length of result string is stored in `int *` placeholder. Caller must free() the result.
- %H: consumes `int *, char **`. Expects a hex-encoded string, e.g. "fa014f". Result string is hex-decoded, malloced and NUL-terminated. The length of the result string is stored in `int *` placeholder. Caller must free() the result.
- %M: consumes custom scanning function pointer and `void *user_data` parameter - see `json_scanner_t` definition.
- %T: consumes `struct json_token *`, fills it out with matched token.

Returns the number of elements successfully scanned & converted.

Negative number means scan error.

Example - scan arbitrary JSON string:

```

// str has the following JSON string (notice keys are out of order):
// { "a": 123, "d": true, "b": [1, 2], "c": "hi" }

int a = 0, d = 0;
char *c = NULL;
void *my_data = NULL;
json_scanf(str, strlen(str), "{ a:%d, b:%M, c:%Q, d:%B }",
           &a, scan_array, my_data, &c, &d);

// This function is called by json_scanf() call above.
// str is "[1, 2]", user_data is my_data.
static void scan_array(const char *str, int len, void *user_data) {
    struct json_token t;
    int i;
    printf("Parsing array: %.*s\n", len, str);
    for (i = 0; json_scanf_array_elem(str, len, "", i, &t) > 0; i++) {
        printf("Index %d, token [%.*s]\n", i, t.len, t.ptr);
    }
}

```

Example - parse array of objects:

```

// str has the following JSON string - array of objects:
// { "a": [ {"b": 123}, {"b": 345} ] }
// This example shows how to iterate over array, and parse each object.

int i, value, len = strlen(str);
struct json_token t;

for (i = 0; json_scanf_array_elem(str, len, ".a", i, &t) > 0; i++) {
    // t.type == JSON_TYPE_OBJECT
    json_scanf(t.ptr, t.len, "{b: %d}", &value); // value is 123, then 345
}

```

json_scanf_array_elem()

```

int json_scanf_array_elem(const char *s, int len,
                          const char *path,
                          int index,
                          struct json_token *token);

```

A helper function to scan an array item with given path and index.

Fills `token` with the matched JSON token.

Returns 0 if no array element found, otherwise non-0.

json_printf()

Frozen printing API is pluggable. Out of the box, Frozen provides a way to print to a string buffer or to an opened file stream. It is easy to tell Frozen to print to another destination, for example, to a socket, etc. Frozen does this by defining an "output context" descriptor which has a pointer to a low-level printing function. If you want to print to another destination, just define your specific printing function and initialise output context with it.

This is the definition of the output context descriptor:

```

struct json_out {
    int (*printer)(struct json_out *, const char *str, size_t len);
    union {
        struct {
            char *buf;
            size_t size;
            size_t len;
        } buf;
        void *data;
        FILE *fp;
    } u;
};

```

Frozen provides two helper macros to initialise two built-in output descriptors:

```

struct json_out out1 = JSON_OUT_BUF(buf, len);
struct json_out out2 = JSON_OUT_FILE(fp);

```

```

typedef int (*json_printf_callback_t)(struct json_out *, va_list *ap);
int json_printf(struct json_out *, const char *fmt, ...);
int json_vprintf(struct json_out *, const char *fmt, va_list ap);

```

Generate formatted output into a given string buffer, auto-escaping keys.

This is a superset of printf() function, with extra format specifiers:

- %B print json boolean, true or false. Accepts an int.
- %Q print quoted escaped string or null. Accepts a const char *.
- %.*Q same as %Q, but with length. Accepts int, const char *
- %V print quoted base64-encoded string. Accepts a const char *, int.
- %H print quoted hex-encoded string. Accepts a int, const char *.
- %M invokes a json_printf_callback_t function. That callback function can consume more parameters.

Return number of bytes printed. If the return value is bigger than the supplied buffer, that is an indicator of overflow. In the overflow case, overflow bytes are not printed.

Example:

```

json_printf(&out, "{%Q: %d, x: [%B, %B], y: %Q}", "foo", 123, 0, -1, "hi");
// Result:
// {"foo": 123, "x": [false, true], "y": "hi"}

```

To print a complex object (for example, serialise a structure into an object), use %M format specifier:

```

struct my_struct { int a, b; } mys = {1,2};
json_printf(&out, "{foo: %M, bar: %d}", print_my_struct, &mys, 3);
// Result:
// {"foo": {"a": 1, "b": 2}, "bar": 3}

```

```

int print_my_struct(struct json_out *out, va_list *ap) {
    struct my_struct *p = va_arg(*ap, struct my_struct *);
    return json_printf(out, "{a: %d, b: %d}", p->a, p->b);
}

```

json_printf_array()

```
int json_printf_array(struct json_out *, va_list *ap);
```

A helper %M callback that prints contiguous C arrays.

Consumes void *array_ptr, size_t array_size, size_t elem_size, char *fmt

Returns number of bytes printed.

json_walk() - low level parsing API

```
/* JSON token type */
enum json_token_type {
    JSON_TYPE_INVALID = 0, /* memsetting to 0 should create INVALID value */
    JSON_TYPE_STRING,
    JSON_TYPE_NUMBER,
    JSON_TYPE_TRUE,
    JSON_TYPE_FALSE,
    JSON_TYPE_NULL,
    JSON_TYPE_OBJECT_START,
    JSON_TYPE_OBJECT_END,
    JSON_TYPE_ARRAY_START,
    JSON_TYPE_ARRAY_END,
    JSON_TYPES_CNT,
};

/*
 * Structure containing token type and value. Used in `json_walk()` and
 * `json_scanf()` with the format specifier `%T` .
 */
struct json_token {
    const char *ptr;           /* Points to the beginning of the value */
    int len;                  /* Value length */
    enum json_token_type type; /* Type of the token, possible values are above */
};

/* Callback-based API */
typedef void (*json_walk_callback_t)(void *callback_data,
                                      const char *name, size_t name_len,
                                      const char *path,
                                      const struct json_token *token);

/*
 * Parse `json_string` , invoking `callback` in a way similar to SAX parsers;
 * see `json_walk_callback_t` .
 */
int json_walk(const char *json_string, int json_string_length,
              json_walk_callback_t callback, void *callback_data);
```

json_walk() is a low-level, callback based parsing API.

json_walk() calls a given callback function for each scanned value.

Callback receives a name, a path to the value, a JSON token that points to the value and an arbitrary user data pointer.

The path is constructed using this rule:

- Root element has "" (empty string) path
- When an object starts, . (dot) is appended to the path
- When an object key is parsed, a key name is appended to the path
- When an array is parsed, an [ELEMENT_INDEX] is appended for each element

For example, consider the following json string:

```
{ "foo": 123, "bar": [ 1, 2, { "baz": true } ] }.
```

The sequence of callback invocations will be as follows:

- type: JSON_TYPE_OBJECT_START, name: NULL, path: "", value: NULL
- type: JSON_TYPE_NUMBER, name: "foo", path: ".foo", value: "123"
- type: JSON_TYPE_ARRAY_START, name: "bar", path: ".bar", value: NULL
- type: JSON_TYPE_NUMBER, name: "0", path: ".bar[0]", value: "1"
- type: JSON_TYPE_NUMBER, name: "1", path: ".bar[1]", value: "2"
- type: JSON_TYPE_OBJECT_START, name: "2", path: ".bar[2]", value: NULL
- type: JSON_TYPE_TRUE, name: "baz", path: ".bar[2].baz", value: "true"
- type: JSON_TYPE_OBJECT_END, name: NULL, path: ".bar[2]", value: "{ \"baz\": true }"
- type: JSON_TYPE_ARRAY_END, name: NULL, path: ".bar", value: "[1, 2, { \"baz\": true }]"
- type: JSON_TYPE_OBJECT_END, name: NULL, path: "", value: "{ \"foo\": 123, \"bar\": [1, 2, { \"baz\": true }] }"

If top-level element is an array: [1, {"foo": 2}]

- type: JSON_TYPE_ARRAY_START, name: NULL, path: "", value: NULL
- type: JSON_TYPE_NUMBER, name: "0", path: "[0]", value: "1"
- type: JSON_TYPE_OBJECT_START, name: "1", path: "[1]", value: NULL
- type: JSON_TYPE_NUMBER, name: "foo", path: "[1].foo", value: "2"
- type: JSON_TYPE_OBJECT_END, name: NULL, path: "[1]", value: "{\"foo\": 2}"
- type: JSON_TYPE_ARRAY_END, name: NULL, path: "", value: "[1, {\"foo\": 2}]"

If top-level element is a scalar: true

- type: JSON_TYPE_TRUE, name: NULL, path: "", value: "true"

json_fprintf(), json_vfprintf()

```
/*
 * Same as json_printf, but prints to a file.
 * File is created if does not exist. File is truncated if already exists.
 */
int json_fprintf(const char *file_name, const char *fmt, ...);
int json_vfprintf(const char *file_name, const char *fmt, va_list ap);
```

json_asprintf(), json_vasprintf()

```
/*
 * Print JSON into an allocated 0-terminated string.
 * Return allocated string, or NULL on error.
 * Example:
 *
 * ````c
 *     char *str = json_asprintf("{a:%H}", 3, "abc");
 *     printf("%s\n", str); // Prints "616263"
 *     free(str);
 * ````
 */
char *json_asprintf(const char *fmt, ...);
char *json_vasprintf(const char *fmt, va_list ap);
```

json_fread()

```
/*
 * Read the whole file in memory.
 * Return malloc-ed file content, or NULL on error. The caller must free().
 */
char *json_fread(const char *file_name);
```

json_setf(), json_vsetf()

```
/*
 * Update given JSON string `s,len` by changing the value at given `json_path` .
 * The result is saved to `out` . If `json_fmt` == NULL, that deletes the key.
 * If path is not present, missing keys are added. Array path without an
 * index pushes a value to the end of an array.
 * Return 1 if the string was changed, 0 otherwise.
 *
 * Example: s is a JSON string { "a": 1, "b": [ 2 ] }
 *   json_setf(s, len, out, ".a", "7");      // { "a": 7, "b": [ 2 ] }
 *   json_setf(s, len, out, ".b", "7");      // { "a": 1, "b": 7 }
 *   json_setf(s, len, out, ".b[]", "7");    // { "a": 1, "b": [ 2,7 ] }
 *   json_setf(s, len, out, ".b", NULL);     // { "a": 1 }
 */
int json_setf(const char *s, int len, struct json_out *out,
              const char *json_path, const char *json_fmt, ...);

int json_vsetf(const char *s, int len, struct json_out *out,
               const char *json_path, const char *json_fmt, va_list ap);
```

json_prettyify()

```
/*
 * Pretty-print JSON string `s,len` into `out` .
 * Return number of processed bytes in `s` .
 */
int json_prettyify(const char *s, int len, struct json_out *out);
```

json_prettyify_file()

```
/*
 * Prettify JSON file `file_name` .
 * Return number of processed bytes, or negative number of error.
 * On error, file content is not modified.
 */
int json_prettyify_file(const char *file_name);
```

json_next_key(), json_next_elem()

```

/*
 * Iterate over an object at given JSON `path`.
 * On each iteration, fill the `key` and `val` tokens. It is OK to pass NULL
 * for `key` , or `val` , in which case they won't be populated.
 * Return an opaque value suitable for the next iteration, or NULL when done.
 *
 * Example:
 *
 * ````c
 * void *h = NULL;
 * struct json_token key, val;
 * while ((h = json_next_key(s, len, h, ".foo", &key, &val)) != NULL) {
 *     printf("[%.*s] -> [%.*s]\n", key.len, key.ptr, val.len, val.ptr);
 * }
 * ```
 */
void *json_next_key(const char *s, int len, void *handle, const char *path,
                    struct json_token *key, struct json_token *val);

/*
 * Iterate over an array at given JSON `path` .
 * Similar to `json_next_key` , but fills array index `idx` instead of `key` .
 */
void *json_next_elem(const char *s, int len, void *handle, const char *path,
                     int *idx, struct json_token *val);

```

Minimal mode

By building with `-DJSON_MINIMAL=1` footprint can be significantly reduced.

The following limitations apply in this configuration:

- Only integer numbers are supported. This affects parsing and `%f/%lf` conversions in `printf` and `scanf`.
- Hex ('%H') and base64 (%V) conversions are disabled.

Examples

Print JSON configuration to a file

```

json_fprintf("settings.json", "{ a: %d, b: %Q }", 123, "string_value");
json_prettyfile("settings.json"); // Optional

```

Read JSON configuration from a file

```

struct my_config { int a; char *b; } c = { .a = 0, .b = NULL };
char *content = json_fread("settings.json");
json_scanf(content, strlen(content), "{a: %d, b: %Q}", &c.a, &c.b);

```

Modify configuration setting in a JSON file

```
const char *settings_file_name = "settings.json", *tmp_file_name = "tmp.json";
char *content = json_fread(settings_file_name);
FILE *fp = fopen(tmp_file_name, "w");
struct json_out out = JSON_OUT_FILE(fp);
json_setf(content, strlen(content), &out, ".b", "%Q", "new_string_value");
fclose(fp);
json_prettyify_file(tmp_file_name); // Optional
rename(tmp_file_name, settings_file_name);
```

Contributions

To submit contributions, sign

[Cesanta CLA](#)

and send GitHub pull request.

Licensing

Frozen is released under the

[Apache 2.0 license](#).

**For commercial support and professional services,
contact us.**

JSON parser and emitter for C/C++

Features

- ISO C and ISO C++ compliant portable code
- Very small footprint
- No dependencies
- `json_scanf()` scans a string directly into C/C++ variables
- `json_printf()` prints C/C++ variables directly into an output stream
- `json_setf()` modifies an existing JSON string
- `json_fread()` reads JSON from a file
- `json_fprintf()` writes JSON to a file
- Built-in base64 encoder and decoder for binary data
- Parser provides low-level callback API and high-level scanf-like API
- 100% test coverage
- Used in [Mongoose OS](#), an operating system
for connected commercial products on low-power microcontrollers

API reference

json_scanf(), json_vscanf

```

int json_scanf(const char *str, int str_len, const char *fmt, ...);
int json_vscanf(const char *str, int str_len, const char *fmt, va_list ap);

/* json_scanf's %M handler */
typedef void (*json_scanner_t)(const char *str, int len, void *user_data);

```

Scans the JSON string `str`, performing scanf-like conversions according to `fmt`.

`fmt` uses `scanf()`-like format, with the following differences:

1. Object keys in the format string don't have to be quoted, e.g. "{key: %d}"
2. Order of keys in the format string does not matter, and the format string may omit keys to fetch only those that are of interest, for example, assume `str` is a JSON string `{ "a": 123, "b": "hi", c: true }`.

We can fetch only the value of the `c` key:

```

int value = 0;
json_scanf(str, strlen(str), "{c: %B}", &value);

```

3. Several extra format specifiers are supported:

- %B: consumes `int *` (or `char *`, if `sizeof(bool) == sizeof(char)`), expects boolean `true` or `false`.
- %Q: consumes `char **`, expects quoted, JSON-encoded string. Scanned string is malloc-ed, caller must free() the string.
- %V: consumes `char **, int *`. Expects base64-encoded string.
Result string is base64-decoded, malloced and NUL-terminated.
The length of result string is stored in `int *` placeholder.
Caller must free() the result.
- %H: consumes `int *, char **`.
Expects a hex-encoded string, e.g. "fa014f".
Result string is hex-decoded, malloced and NUL-terminated.
The length of the result string is stored in `int *` placeholder.
Caller must free() the result.
- %M: consumes custom scanning function pointer and `void *user_data` parameter - see `json_scanner_t` definition.
- %T: consumes `struct json_token *`, fills it out with matched token.

Returns the number of elements successfully scanned & converted.

Negative number means scan error.

Example - scan arbitrary JSON string:

```

// str has the following JSON string (notice keys are out of order):
// { "a": 123, "d": true, "b": [1, 2], "c": "hi" }

int a = 0, d = 0;
char *c = NULL;
void *my_data = NULL;
json_scanf(str, strlen(str), "{ a:%d, b:%M, c:%Q, d:%B }",
           &a, scan_array, my_data, &c, &d);

// This function is called by json_scanf() call above.
// str is "[1, 2]", user_data is my_data.
static void scan_array(const char *str, int len, void *user_data) {
    struct json_token t;
    int i;
    printf("Parsing array: %.s\n", len, str);
    for (i = 0; json_scan_array_elem(str, len, "", i, &t) > 0; i++) {
        printf("Index %d, token [%.*s]\n", i, t.len, t.ptr);
    }
}

```

Example - parse array of objects:

```
// str has the following JSON string - array of objects:  
// { "a": [ {"b": 123}, {"b": 345} ] }  
// This example shows how to iterate over array, and parse each object.  
  
int i, value, len = strlen(str);  
struct json_token t;  
  
for (i = 0; json_scanf_array_elem(str, len, ".a", i, &t) > 0; i++) {  
    // t.type == JSON_TYPE_OBJECT  
    json_scanf(t.ptr, t.len, "{b: %d}", &value); // value is 123, then 345  
}
```

json_scanf_array_elem()

```
int json_scanf_array_elem(const char *s, int len,  
                         const char *path,  
                         int index,  
                         struct json_token *token);
```

A helper function to scan an array item with given path and index.

Fills `token` with the matched JSON token.

Returns 0 if no array element found, otherwise non-0.

json_printf()

Frozen printing API is pluggable. Out of the box, Frozen provides a way to print to a string buffer or to an opened file stream. It is easy to tell Frozen to print to another destination, for example, to a socket, etc. Frozen does this by defining an "output context" descriptor which has a pointer to a low-level printing function. If you want to print to another destination, just define your specific printing function and initialise output context with it.

This is the definition of the output context descriptor:

```
struct json_out {  
    int (*printer)(struct json_out *, const char *str, size_t len);  
    union {  
        struct {  
            char *buf;  
            size_t size;  
            size_t len;  
        } buf;  
        void *data;  
        FILE *fp;  
    } u;  
};
```

Frozen provides two helper macros to initialise two built-in output descriptors:

```
struct json_out out1 = JSON_OUT_BUF(buf, len);  
struct json_out out2 = JSON_OUT_FILE(fp);  
  
typedef int (*json_printf_callback_t)(struct json_out *, va_list *ap);  
int json_printf(struct json_out *, const char *fmt, ...);  
int json_vprintf(struct json_out *, const char *fmt, va_list ap);
```

Generate formatted output into a given string buffer, auto-escaping keys.

This is a superset of printf() function, with extra format specifiers:

- %B print json boolean, true or false. Accepts an int.
- %Q print quoted escaped string or null. Accepts a const char *.
- %.*Q same as %Q, but with length. Accepts int, const char *
- %V print quoted base64-encoded string. Accepts a const char *, int.
- %H print quoted hex-encoded string. Accepts a int, const char *.
- %M invokes a json_printf_callback_t function. That callback function can consume more parameters.

Return number of bytes printed. If the return value is bigger than the supplied buffer, that is an indicator of overflow. In the overflow case, overflowed bytes are not printed.

Example:

```
json_printf(&out, "{%Q: %d, x: [%B, %B], y: %Q}", "foo", 123, 0, -1, "hi");
// Result:
// {"foo": 123, "x": [false, true], "y": "hi"}
```

To print a complex object (for example, serialise a structure into an object), use %M format specifier:

```
struct my_struct { int a, b; } mys = {1,2};
json_printf(&out, "{foo: %M, bar: %d}", print_my_struct, &mys, 3);
// Result:
// {"foo": {"a": 1, "b": 2}, "bar": 3}
```

```
int print_my_struct(struct json_out *out, va_list *ap) {
    struct my_struct *p = va_arg(*ap, struct my_struct *);
    return json_printf(out, "{a: %d, b: %d}", p->a, p->b);
}
```

json_printf_array()

```
int json_printf_array(struct json_out *, va_list *ap);
```

A helper %M callback that prints contiguous C arrays.

Consumes void *array_ptr, size_t array_size, size_t elem_size, char *fmt

Returns number of bytes printed.

json_walk() - low level parsing API

```

/* JSON token type */
enum json_token_type {
    JSON_TYPE_INVALID = 0, /* memsetting to 0 should create INVALID value */
    JSON_TYPE_STRING,
    JSON_TYPE_NUMBER,
    JSON_TYPE_TRUE,
    JSON_TYPE_FALSE,
    JSON_TYPE_NULL,
    JSON_TYPE_OBJECT_START,
    JSON_TYPE_OBJECT_END,
    JSON_TYPE_ARRAY_START,
    JSON_TYPE_ARRAY_END,
    JSON_TYPES_CNT,
};

/*
 * Structure containing token type and value. Used in `json_walk()` and
 * `json_scanf()` with the format specifier `%T` .
 */
struct json_token {
    const char *ptr;           /* Points to the beginning of the value */
    int len;                  /* Value length */
    enum json_token_type type; /* Type of the token, possible values are above */
};

/* Callback-based API */
typedef void (*json_walk_callback_t)(void *callback_data,
                                      const char *name, size_t name_len,
                                      const char *path,
                                      const struct json_token *token);

/*
 * Parse `json_string`, invoking `callback` in a way similar to SAX parsers;
 * see `json_walk_callback_t` .
 */
int json_walk(const char *json_string, int json_string_length,
              json_walk_callback_t callback, void *callback_data);

```

`json_walk()` is a low-level, callback based parsing API.

`json_walk()` calls a given callback function for each scanned value.

Callback receives a name, a path to the value, a JSON token that points to the value and an arbitrary user data pointer.

The path is constructed using this rule:

- Root element has "" (empty string) path
- When an object starts, . (dot) is appended to the path
- When an object key is parsed, a key name is appended to the path
- When an array is parsed, an [ELEMENT_INDEX] is appended for each element

For example, consider the following json string:

{ "foo": 123, "bar": [1, 2, { "baz": true }] }.

The sequence of callback invocations will be as follows:

- type: JSON_TYPE_OBJECT_START, name: NULL, path: "", value: NULL
- type: JSON_TYPE_NUMBER, name: "foo", path: ".foo", value: "123"
- type: JSON_TYPE_ARRAY_START, name: "bar", path: ".bar", value: NULL
- type: JSON_TYPE_NUMBER, name: "0", path: ".bar[0]", value: "1"
- type: JSON_TYPE_NUMBER, name: "1", path: ".bar[1]", value: "2"
- type: JSON_TYPE_OBJECT_START, name: "2", path: ".bar[2]", value: NULL
- type: JSON_TYPE_TRUE, name: "baz", path: ".bar[2].baz", value: "true"
- type: JSON_TYPE_OBJECT_END, name: NULL, path: ".bar[2]", value: "{ \"baz\": true }"
- type: JSON_TYPE_ARRAY_END, name: NULL, path: ".bar", value: "[1, 2, { \"baz\": true }]"

- type: JSON_TYPE_OBJECT_END, name: NULL, path: "", value: "{ \"foo\": 123, \"bar\": [1, 2, { \"baz\": true }] }"

If top-level element is an array: [1, {"foo": 2}]

- type: JSON_TYPE_ARRAY_START, name: NULL, path: "", value: NULL
- type: JSON_TYPE_NUMBER, name: "0", path: "[0]", value: "1"
- type: JSON_TYPE_OBJECT_START, name: "1", path: "[1]", value: NULL
- type: JSON_TYPE_NUMBER, name: "foo", path: "[1].foo", value: "2"
- type: JSON_TYPE_OBJECT_END, name: NULL, path: "[1]", value: "{\"foo\": 2}"
- type: JSON_TYPE_ARRAY_END, name: NULL, path: "", value: "[1, {\"foo\": 2}]"

If top-level element is a scalar: true

- type: JSON_TYPE_TRUE, name: NULL, path: "", value: "true"

json_fprintf(), json_vfprintf()

```
/*
 * Same as json_printf, but prints to a file.
 * File is created if does not exist. File is truncated if already exists.
 */
int json_fprintf(const char *file_name, const char *fmt, ...);
int json_vfprintf(const char *file_name, const char *fmt, va_list ap);
```

json_asprintf(), json_vasprintf()

```
/*
 * Print JSON into an allocated 0-terminated string.
 * Return allocated string, or NULL on error.
 * Example:
 *
 *   char *str = json_asprintf("{a:%H}", 3, "abc");
 *   printf("%s\n", str); // Prints "616263"
 *   free(str);
 *
 */
char *json_asprintf(const char *fmt, ...);
char *json_vasprintf(const char *fmt, va_list ap);
```

json_fread()

```
/*
 * Read the whole file in memory.
 * Return malloc-ed file content, or NULL on error. The caller must free().
 */
char *json_fread(const char *file_name);
```

json_setf(), json_vsetf()

```

/*
 * Update given JSON string `s,len` by changing the value at given `json_path` .
 * The result is saved to `out` . If `json_fmt` == NULL, that deletes the key.
 * If path is not present, missing keys are added. Array path without an
 * index pushes a value to the end of an array.
 * Return 1 if the string was changed, 0 otherwise.
 *
 * Example: s is a JSON string { "a": 1, "b": [ 2 ] }
 *   json_setf(s, len, out, ".a", "7");      // { "a": 7, "b": [ 2 ] }
 *   json_setf(s, len, out, ".b", "7");      // { "a": 1, "b": 7 }
 *   json_setf(s, len, out, ".b[]", "7");    // { "a": 1, "b": [ 2,7 ] }
 *   json_setf(s, len, out, ".b", NULL);     // { "a": 1 }
 */
int json_setf(const char *s, int len, struct json_out *out,
              const char *json_path, const char *json_fmt, ...);

int json_vsetf(const char *s, int len, struct json_out *out,
               const char *json_path, const char *json_fmt, va_list ap);

```

json_pretty()

```

/*
 * Pretty-print JSON string `s,len` into `out` .
 * Return number of processed bytes in `s` .
 */
int json_prettyf(const char *s, int len, struct json_out *out);

```

json_pretty_file()

```

/*
 * Prettify JSON file `file_name` .
 * Return number of processed bytes, or negative number of error.
 * On error, file content is not modified.
 */
int json_pretty_file(const char *file_name);

```

json_next_key(), json_next_elem()

```

/*
 * Iterate over an object at given JSON `path`.
 * On each iteration, fill the `key` and `val` tokens. It is OK to pass NULL
 * for `key` , or `val` , in which case they won't be populated.
 * Return an opaque value suitable for the next iteration, or NULL when done.
 *
 * Example:
 *
 * ````c
 * void *h = NULL;
 * struct json_token key, val;
 * while ((h = json_next_key(s, len, h, ".foo", &key, &val)) != NULL) {
 *     printf("[%.*s] -> [%.*s]\n", key.len, key.ptr, val.len, val.ptr);
 * }
 * ```
 */
void *json_next_key(const char *s, int len, void *handle, const char *path,
                    struct json_token *key, struct json_token *val);

/*
 * Iterate over an array at given JSON `path` .
 * Similar to `json_next_key` , but fills array index `idx` instead of `key` .
 */
void *json_next_elem(const char *s, int len, void *handle, const char *path,
                     int *idx, struct json_token *val);

```

Minimal mode

By building with `-DJSON_MINIMAL=1` footprint can be significantly reduced.

The following limitations apply in this configuration:

- Only integer numbers are supported. This affects parsing and `%f/%lf` conversions in `printf` and `scanf`.
- Hex ('%H') and base64 (%V) conversions are disabled.

Examples

Print JSON configuration to a file

```

json_fprintf("settings.json", "{ a: %d, b: %Q }", 123, "string_value");
json_prettyfile("settings.json"); // Optional

```

Read JSON configuration from a file

```

struct my_config { int a; char *b; } c = { .a = 0, .b = NULL };
char *content = json_fread("settings.json");
json_scanf(content, strlen(content), "{a: %d, b: %Q}", &c.a, &c.b);

```

Modify configuration setting in a JSON file

```
const char *settings_file_name = "settings.json", *tmp_file_name = "tmp.json";
char *content = json_fread(settings_file_name);
FILE *fp = fopen(tmp_file_name, "w");
struct json_out out = JSON_OUT_FILE(fp);
json_setf(content, strlen(content), &out, ".b", "%Q", "new_string_value");
fclose(fp);
json_prettyify_file(tmp_file_name); // Optional
rename(tmp_file_name, settings_file_name);
```

Contributions

To submit contributions, sign

[Cesanta CLA](#)

and send GitHub pull request.

Licensing

Frozen is released under the

[Apache 2.0 license](#).

For commercial support and professional services,
[contact us](#).

Mongoose OS and Microsoft Azure IoT

This tutorial demonstrates the following:

- How to build a skeleton for a commercial smart device, managed using Microsoft Azure IoT
- How to reuse the code for different hardware platforms
- How to configure devices via Azure IoT Hub device twin
- How to perform bulk over-the-air (OTA) updates using [Azure IoT Hub automatic device management](#)

Setup Azure IoT Hub

- Create an account on [portal.azure.com](#)
- Install `az`, the Azure command line utility, by [following these instructions](#)
- Start command prompt (or terminal on UNIX/Mac), login and follow the instructions:

```
az login
```

- Create an IoT hub. Note:
 - Use EU-West region (this is important, since not all regions have IoT device configuration enabled)
 - Choose standard pricing tier

Setup device

- Pick one of the supported devices. We suggest to choose from [recommended devboards](#)
- Connect your device to your workstation via USB
- Complete [Quickstart Guide](#) steps 1-7 inclusive.
As a result, your device should be connected to the Internet
- Provision your device to Azure IoT with a single command:

```
mos azure-iot-setup --azure-hub-name YOUR_AZURE_HUB_NAME
```

This command performs certificate management for you, and onboards your device on a cloud. If your device has an [Atmel ECC508A](#) secure element attached, Mongoose OS automatically uses ECC508A chip for TLS and keeps your credentials securely stored.

A newly provisioned device will appear in the IoT hub's device list. On the picture, an ESP8266 board was used. You will get a different device ID, depending on the hardware platform you're using.

Configuring the device using Azure IoT Hub device twin

In the IoT Hub blade of the Azure portal, click on the device ID shown in the device list, then select the "Device twin" tab:

That will bring up the device twin editor:

The firmware that we've build following quickstart guide, contains

code that synchronises device LED with the `desired.on` twin setting.

In the device twin editor, add `desired.on` boolean key:

```
...
"desired": {
  "on": true,    // <-- add this
  ...
},
...
```

Set it to `true` or

`false`, and observe how the device reacts by switching the LED on/off:



This example demonstrates remote device configuration using the device twin, and can be applied to a broad range of use cases.

IoT button

Open `fs/init.js` file in your favorite editor, copy/paste the following code:

```

load('api_config.js');
load('api_gpio.js');
load('api_sys.js');
load('api_mqtt.js');

let pin = Cfg.get('board.btn1.pin'); // User button GPIO pin
let topic = 'devices/' + Cfg.get('device.id') + '/messages/events/';

let f = function() {
  let message = JSON.stringify({
    total_ram: Sys.total_ram(),
    free_ram: Sys.free_ram(),
    uptime: Sys.uptime(),
  });
  let ok = MQTT.pub(topic, message, 1);
  print('Published:', ok, topic, '->', message);
};

GPIO.set_button_handler(pin, GPIO.PULL_UP, GPIO.INT_EDGE_NEG, 20, f, null);

```

When done, copy `fs/init.js` to the device and reboot the device:

```

mos put fs/init.js
mos call Sys.Reboot

```

Then, press a button to send an MQTT message.

Bulk OTA updates with the Azure IoT Hub automatic device management feature

Register a couple more devices in the Azure IoT Hub, following the same steps as in the Setup Device section above:

```
mos azure-iot-setup --azure-hub-name YOUR_AZURE_HUB_NAME
```

For each device, edit the Device Twin adding the following snippet:

```

"tags": {
  "city": "dublin"
},

```

Make some change in the firmware code. Replace the code in `fs/init.js` with
`print('hello new version');` and rebuild the firmware:

```

mos build --platform YOUR_PLATFORM
mos console

```

This will generate a new firmware `build/fw.zip`. Upload this file to some web server of your choice.

In the Azure IoT Hub blade go to the Automatic Device management section and create a new device configuration:

Give this configuration a name. On step 2, specify twin settings you'd like to apply: "Device Twin Path" set to `properties.desired`, and "Content" to

```

{
  "ota": {
    "url": "URL_OF_THE_REBUILT_fw.zip"
  }
}

```

Skip Step 3, and on Step 4 specify "Target Condition":

`tags.city='dublin'`. Submit configuration.

Open the device twin of any device, make sure that the `desired.properties.ota.url` attribute has been applied. The firmware we've built uses the [ota-shadow](#) library which observes that attribute in the twin, and starts an OTA update if it is changes.

The OTA parameters are reported by the firmware in the `properties.reported.ota`, where you can see various metrics:

```
"reported": {  
  "ota": {  
    "fw_version": "1.0",  
    "fw_id": "20180716-150324/lsm@23146fe+",  
    "mac": "5ECF7F060046",  
    "device_id": "esp8266_060046",  
    "app": "demo-js",  
    "message": "progress",  
    "status": 1,  
    "is_committed": true,  
    "commit_timeout": 600,  
    "partition": 0,  
    "progress_percent": 27,  
    ...  
  },
```

Note the `progress_percent` value. If you refresh the twin editor, it'll change to indicate the OTA update progress. When the update is complete, the value of `ota.fw_id` changes to indicate the new timestamp of the firmware.

Observe console statement, note the `Hello new version` message.

You have just performed an over-the-air firmware update accross multiple devices seamlessly leveraging Azure IoT.

Mongoose OS + private MQTT server

The `mqtt` library provides [MQTT protocol](#) API that allows devices to talk to MQTT servers.

Mongoose OS implements MQTT 3.1.1 client functionality, and works with all popular MQTT server implementations, like AWS IoT, Google IoT Core, Microsoft Azure, IBM Watson, HiveMQ, Mosquitto, etc.

Usage

In order to use MQTT functionality,

- Include `mqtt` library in the `libs` section of the `mos.yml` file of your app:

```
libs:  
  ...  
  - origin: https://github.com/mongoose-os-libs/mqtt # <-- Add this!
```

- Rebuild and reflash your app:

```
mos build --platform YOUR_PLATFORM # e.g. stm32, esp32, cc3220, esp8266, cc3200, etc  
mos flash
```

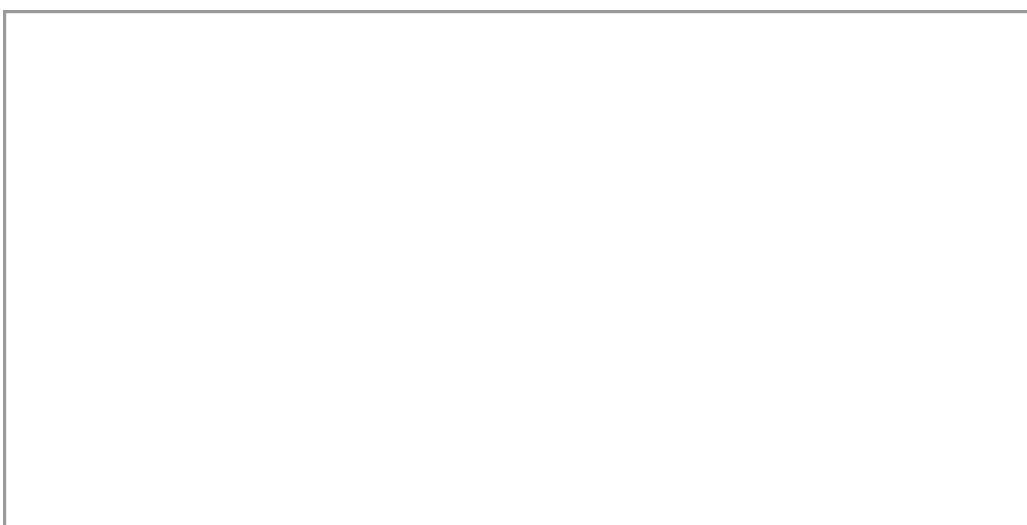
NOTE: if you're using a pre-built Mongoose OS demo-js app, that step is not required, since mqtt library is already included in the demo-js.

In order to talk to an MQTT server, configure MQTT server settings - see Configuration section below. Once configured, Mongoose OS keeps that connection alive by reconnecting and re-subscribing to all topics after disconnections - you do not need to implement the reconnection logic.

```
$ mos config-set mqtt.enable=true mqtt.server=my_server:1883
```

If you want to use TLS, set mqtt.ssl_ca_cert=ca.pem. Make sure that ca.pem file has required CA certificates. If you want to use mutual TLS, set mqtt.ssl_cert=CLIENT_CERT.pem and mqtt.ssl_key=PRIVATE_KEY.pem.

See example video (don't forget to set mqtt.enable=true before you try it):



MQTT publish and subscribe - JavaScript

```
load('api_mqtt.js');  
load('api_gpio.js');  
  
let pin = 0, topic = 'my/topic';  
  
GPIO.set_button_handler(pin, GPIO.PULL_UP, GPIO.INT_EDGE_NEG, 200, function() {  
  MQTT.pub('my/topic', JSON.stringify({ a: 1, b: 2 }));  
}, null);  
  
MQTT.sub('my/topic/#', function(conn, topic, msg) {  
  print('Topic:', topic, 'message:', msg);  
}, null);
```

MQTT publish and subscribe - C/C++

```

# include "mgos_mqtt.h"
<p class="mume-header" id="include-mqtth"></p>

static void handler(struct mg_connection *c, const char *topic, int topic_len,
                    const char *msg, int msg_len, void *userdata) {
    LOG(LL_INFO, ("Got message on topic %.%s", topic_len, topic));
    ...
}

mgos_mqtt_sub("my/#", handler, NULL);      /* Subscribe */
mgos_mqtt_pub("my/topic", "hi", 2, 1, 0);   /* Publish */

```

Configuration Reference

The MQTT library adds `mqtt` section to the device configuration:

```
{
  "clean_session": true,           // Clean session info stored on server
  "client_id": "",                // If not set, device.id is used
  "enable": false,                 // Enable MQTT functionality
  "keep_alive": 60,                // How often to send PING messages in seconds
  "pass": "",                     // User password
  "reconnect_timeout_max": 60,     // Maximum reconnection timeout in seconds
  "reconnect_timeout_min": 2,       // Minimum reconnection timeout in seconds
  "server": "iot.eclipse.org:1883", // SERVER:PORT to connect to
  "ssl_ca_cert": "",              // Set this to file name with CA certs to enable TLS
  "ssl_cert": "",                 // Client certificate for mutual TLS
  "ssl_cipher_suites": "",        // TLS cipher suites
  "ssl_key": "",                  // Private key for the client certificate
  "ssl_psk_identity": "",         // If set, a preshared key auth is used
  "ssl_psk_key": "",              // Preshared key
  "user": "",                     // MQTT user name, if MQTT auth is used
  "will_message": "",             // MQTT last will message
  "will_topic": ""                // MQTT last will topic
}
```

Mongoose OS + Google IoT Core

This tutorial demonstrates the following:

- How to build a skeleton firmware for a commercial smart device, managed by Google IoT Core
- How to reuse the code for different hardware platforms
- How to control devices via Google IoT Core `config` object
- How to report device state via Google IoT Core `state` object

See following video that repeats setup steps mentioned below:



Setup Google IoT Core

- Install [gcloud command line tool](#)
- Authenticate with Google Cloud:

```
gcloud auth application-default login
```

- Create cloud project - choose your unique project name:

```
gcloud projects create YOUR_PROJECT_NAME
```

- Add permissions for IoT Core:

```
gcloud projects add-iam-policy-binding YOUR_PROJECT_NAME --member=serviceAccount:cloud-iot@system.gserviceaccount.com --role=roles/pubsub.publish
```

- Set default values for gcloud:

```
gcloud config set project YOUR_PROJECT_NAME
```

- Create PubSub topic for device data:

```
gcloud pubsub topics create iot-topic
```

- Create PubSub subscription for device data:

```
gcloud pubsub subscriptions create --topic iot-topic iot-subscription
```

- Create device registry:

```
gcloud iot registries create iot-registry --region europe-west1 --event-notification-config=topic=iot-topic
```

Setup device

- Pick one of the supported devices. We suggest to choose from [recommended devboards](#)
- Connect your device to your workstation via USB
- Complete [Quickstart Guide](#) steps 1-7 inclusive.
As a result, your device should be connected to the Internet
- Get project ID of your new project:

```
gcloud projects list
```

- Register device on Google IoT Core. If a device is already registered, this command deletes it, then registers again. Note that this command is using `YOUR_PROJECT_ID` instead of `YOUR_PROJECT_NAME`. Take the project ID from the result of your previous command:

```
mos gcp-iot-setup --gcp-project YOUR_PROJECT_ID --gcp-region europe-west1 --gcp-registry iot-registry
```

This command performs certificate management for you, and onboard your device on a cloud. If your device has an [Atmel ECC508A](#) secure element attached, Mongoose OS automatically uses ECC508A chip for TLS and keeps your credentials securely stored.

Controlling LED via Google IoT Core

Open `fs/init.js` in your favorite editor, copy-paste the following snippet and save:

```
load('api_config.js');
load('api_gpio.js');
load('api_mqtt.js');

let led = Cfg.get('board.led1.pin');
let topic = '/devices/' + Cfg.get('device.id') + '/config';

GPIO.set_mode(led, GPIO.MODE_OUTPUT);

MQTT.sub(topic, function(conn, topic, msg) {
  print('Topic:', topic, 'message:', msg);
  let obj = JSON.parse(msg) || {led: 0};
  GPIO.write(led, obj.led);
}, null);
```

The snippet above subscribes to the `/config` notifications. Once the config object is sent to device by Google IoT Core, the handler function is called and lights the corresponding LED.

In the command prompt (or terminal on Linux/Mac), enter the following commands to copy `init.js` to the device, reboot the device, and start monitoring serial output:

```
mos put fs/init.js
mos call Sys.Reboot
mos console
```



Find a registered device in a Google IoT Core device registry. Click on "Update Config" button, and in the opened dialog, enter the following configuration and hit "SEND TO DEVICE":

```
{"led": 1}
```

Note how one of the LEDs turns on. NOTE: some dev boards, notably Espressif, have LED pins inverted, i.e. they switch off on value 1 and switch on, on value 0. Enter

```
{"led": 0}
```

Note how the LED turns off.

Important thing to note: Google IoT Core send device config object each time device comes online or a config change is made. That means, that if you power cycle the device, it'll sync its LED state with the cloud as soon as it gets connected.

Reporting state to Google IoT Core

We are going to report free RAM to the Google IoT Core periodically, to the Google's special /state object.

Open fs/init.js in your favorite editor, copy-paste the following snippet and save:

```
load('api_config.js');
load('api_mqtt.js');
load('api_sys.js');
load('api_timer.js');

let topic = '/devices/' + Cfg.get('device.id') + '/state';

Timer.set(3000 /* milliseconds */, Timer.REPEAT, function() {
  let msg = JSON.stringify({free: Sys.free_ram(), total: Sys.total_ram()});
  print(topic, '->', msg);
  MQTT.pub(topic, msg, 1);
}, null);
```



In the command prompt (or terminal on Linux/Mac), enter the following commands to copy init.js to the device, reboot the device, and start monitoring serial output:

```
mos put fs/init.js
mos call Sys.Reboot
mos console
```

In the Google IoT Web console, click on the "Configuration and state history", and see how new state objects are arriving. Click on any state object, select "text" view, and check the JSON string that is sent by device.

Using ATECC508A crypto chip

See <https://mongoose-os.com/blog/mongoose-os-google-iot-ecc508a/> on how to use ATECC508A crypto chip with Mongoose OS and Google IoT Core.

Sending debug logs to stackdriver

It may be very useful for remote debugging to be able to see a device's logs in Stackdriver. To do so you need to configure the debug.stdout_topic / debug.stderr_topic fields, add a topic in Google IoT, create a cloud function that listens to the topic and writes a new log entry that you can watch live in Stackdriver. We'll be using the topic name of 'debug', and associating both debug and error logs with it, but you can use whatever you like!

*** Warning *** This may incur a *lot* of storage and processing costs running full debug through the cloud functions and storage. Proceed at own risk.

Configure device

It's recommended to set it via a config{7/8/9}.json file that you load onto the device, or you can simply configure the device via mos tool:

```
mos call Config.Set '{"config": {"debug": {"stdout_topic": "/devices/{YOURDEVICEID}/events/debug", "stderr_topic": "/devices/{YOURDEVICEID}/events/debug"}}}'
```

Replace {YOURDEVICEID} with the ID of the device.

Add a topic

- Log into your cloud console, navigate to IoT core and choose your registry.
- Click add/edit topics
- There's an add topic and subfolder button, click that
- Choose 'add topic' from the popup
- Fill the topic and subfolder fields with 'debug'
- Go back to the add/edit topics screen, scroll down and click update

Create a Firebase cloud function

We presume you know how to use firebase cloud functions, and we'll be assuming you have a local functions folder that you're editing and uploading. Here's some docu if you need to get up to speed: <https://firebase.google.com/docs/functions/get-started>

This is our function to intercept the debug topic and write a meaningful log result to Stackdriver.

```

var functions = require('firebase-functions');
const {Logging} = require('@google-cloud/logging');

// create the Stackdriver Logging client
const logging = new Logging({
    projectId: process.env.GCLOUD_PROJECT,
});

// start cloud function

exports.deviceLog =
    functions.pubsub.topic('debug').onPublish((message) => {

        //console.log("message received: ", message);

        const log = logging.log('debug-logs');
        const metadata = {
            // Set the Cloud IoT Device you are writing a log for
            // you extract the required device info from the PubSub attributes
            resource: {
                type: 'cloudiot_device',
                labels: {
                    project_id: message.attributes.projectId,
                    device_num_id: message.attributes.deviceNumId,
                    device_registry_id: message.attributes.deviceRegistryId,
                    location: message.attributes.location,
                }
            },
            labels: {
                // note device_id is not part of the monitored resource, but you can
                // include it as another log label
                device_id: message.attributes.deviceId,
            }
        };
        });

        let buff = new Buffer(message.data, 'base64');
        let text = buff.toString('ascii');
        //console.log("data: ", text);

        let textArray = text.split("|");
        if (textArray[0].substr(-1) === "2"){
            metadata['severity'] = 'ERROR';
        } else {
            metadata['severity'] = 'DEBUG';
        }

        // write the log entry to Stackdriver Logging
        const entry = log.entry(metadata, textArray[1]);
        return log.write(entry);
    });
}

```

Deploy that function and then go to the Stackdriver console, and drill down to the function to check that it's executing without errors.

View the logs

Within Stackdriver click the resource filter and choose Cloud IoT Device, and you can either choose all device num or a specific (Google ID) device to see the nicely formatted debug logs.

Samsung Artik Cloud

In order to use Samsung Artik Cloud service, a "Generic MQTT server" library is required. The sequence of steps to onboard with Artik Cloud is outlined below.

Prerequisites:

- ESP8266 NodeMCU or ESP32 development boards
- Samsung Artik Cloud account

Onboarding sequence

Install Mongoose OS

Run `mos` tool, install Mongoose OS and set up WiFi

Login to Artik Cloud and create a device with type DHT11

Generate device token

Click on the device, generate device token

Configure device

Open device configuration in the Web UI, click on Expert View, and set the following fields (use your specific device ID and token):

- `mqtt.enable=true`
- `mqtt.server=api.artik.cloud:8883`
- `mqtt.user=YOUR_DEVICE_ID`
- `mqtt.pass=YOUR_DEVICE_TOKEN`
- `mqtt.ssl_ca_cert=ca.pem`

Make device report data

Open Device Files tab, click on `init.js` file. Paste this code that sends fake temperature/humidity data to the cloud:

```
load('api_config.js');
load('api_mqtt.js');
load('api_sys.js');
load('api_timer.js');

let sdid = Cfg.get('mqtt.user');
let topic = '/v1.1/messages/' + sdid;

Timer.set(1000, true, function() {
  let data = {temperature:24, humidity:Sys.free_ram()};
  let ok = MQTT.pub(topic, JSON.stringify(data), 1);
  print('Published', ok, JSON.stringify(data));
}, null);
```

Click 'Save + Reboot' button

Graph reported data

Click on CHARTS tab in Artik Cloud console.

Add graph for DHT11 device, humidity - in reality, that reports free RAM.

Summary

The process above shows how to use Samsung Artik Cloud to authorise devices and report data via MQTT.

IBM Watson

This is a quick start tutorial for [IBM Watson IoT Platform](#) for Mongoose OS.

Quick Setup

- Complete [Quickstart Guide](#) steps 1-7 inclusive.
As a result, your device should be connected to the Internet
- Provision your device to IBM Watson:

```
mos watson-iot-setup
```

This does not require any credentials and will set up the device for [QuickStart cloud dashboard](#) connectivity.

With QuickStart dashboard, you will be able to send data from the device at a limited rate but device management (sending commands to the device) is not available.

Note: in order to make a button press to send events,

- Mongoose OS RPC support should be disabled by

```
mos config-set rpc.mqtt.enable=false
```

- A default topic name should be changed. Open `fs/init.js` and edit `set_button_handler()` callback by changing topic name:

```
let topic = 'iot-2/evt/mos/fmt/json';
```

Save, upload to a device using `mos put fs/init.js`, and reboot.

IBM Cloud Account setup

If you already have an IBM Cloud account and a cloud orgaization set up (you can register for a free trial [here](#)) you can use `mos` to add a device to your org.

You will need to obtain app access credentials for `mos` to use:

- In the left side pane og your Watson IoT Platform dashboard select *Apps*
- Click the *Generate API Key* button in the upper right corner
- For description, enter any meaningful text - e.g. "MOS CLI tool" and click *Next*
- For *Role* select *Standard Application*
- Copy the *API Key* and *Authentication Token* and use them on the `mos` command line:

```
$ mos watson-iot-setup --watson-api-key a-bcdefgh-qwertio --watson-api-auth-token "tokengoeshere"
Using port /dev/ttyUSB0
Connecting to the device...
esp8266 1AFE34A5930F running demo-c
Org ID: bcdefgh
Device ID: esp8266_A5930F
Checking device type "mos"...
Creating device "esp8266_A5930F"...

Updating config:
device.id = esp8266_A5930F
watson.client_id = d:bcdefgh:mos:esp8266_A5930F
watson.enable = true
watson.host_name = bcdefgh.messaging.internetofthings.ibmcloud.com
watson.token = SECRET
Setting new configuration...
Saving and rebooting...
```

Note: As with other flags, values can be specified as environment variables: `MOS_WATSON_API_KEY` and `MOS_WATSON_API_AUTH_TOKEN`.

Invoked with no additional flags, as above, `mos` will use device type `mos` (will be created if necessary) and device ID derived from platform and the device's unique ID (usually MAC address) and a random auth token. Additional flags can be provided to override these defaults:

- `--watson-device-type` - specifies device type
- `--watson-device-id` - specifies device id
- `--watson-device-auth-token` - specifies device auth token instead of generating a random one
- `--watson-api-host-name` - use this host name for API calls instead of `ORG_ID.internetofthings.ibmcloud.com`
- `--watson-messaging-host-name` - use this host name for device connections instead of `ORG_ID.messaging.internetofthings.ibmcloud.com`

Mongoose OS + AWS IoT

This tutorial demonstrates the following:

- How to build a skeleton for a commercial smart device, managed by AWS IoT
- Internet Button in 2 minutes
- How to control devices via AWS IoT device shadow
- How to report metrics data via AWS IoT
- How to perform OTA updates using AWS IoT device shadow

Setup AWS IoT

- Install an [AWS CLI command line utility](#)
- Follow the [AWS CLI configuration guide](#) to setup your AWS credentials
- Verify your credentials setup. Start command prompt
(or, terminal on Linux/Mac), enter the following command:

```
$ aws iot list-things
```

It must execute without errors.

Setup device

- Pick one of the supported devices. We suggest to choose from [recommended devboards](#)
- Connect your device to your workstation via USB
- Complete [Quickstart Guide](#) steps 1-7 inclusive.
As a result, your device should be connected to the Internet
- Provision your device to AWS IoT:

```
mos aws-iot-setup --aws-region YOUR_AWS_REGION
```

This command performs certificate management for you, and onboards your device on a cloud. If your device has an

[Atmel ECC508A](#) secure element

attached, Mongoose OS automatically uses ECC508A chip for TLS and keeps your credentials securely stored.

Open AWS console, "IoT Core" service, and find your provisioned device in the list. On the picture, ESP8266 was used.

Internet button in 2 minutes

Check out a video below that shows how to implement Internet button functionality on AWS IoT in less than 2 minutes:



Controlling LED using AWS device shadow

The firmware that we've build following quickstart guide, contains code that synchronises device LED with the `desired.on` shadow setting.

30



On the AWS console, click on the device ID, choose "Shadow" tab. Click on "Edit" to modify shadow, and add the `desired` section:

```
{  
  "desired": {  
    "on": true  
  },  
  ...  
}
```

Click "Save". Click on "Edit" again, change `on` to `false`. Click "Save". See device logs, and see how LED turns on and off. NOTE: on some boards, (notably, Espressif) LED pins are inverted, therefore LED turns on when `on` is `false`, and turns off when `on` is `true`.

Reporting metrics to AWS IoT

We'll report free RAM metric using MQTT API. Note that AWS IoT has versatile rules engine, which allows to handle incoming MQTT messages in a various ways: store in Dynamo DB, forward to your Lambda function, etc etc.

Open `fs/init.js` file in your favorite editor and copy/paste the following:

```
load('api_config.js');
load('api_mqtt.js');
load('api_sys.js');
load('api_timer.js');

let topic = 'metrics/' + Cfg.get('device.id');

Timer.set(1000 /* milliseconds */, Timer.REPEAT, function() {
  let msg = JSON.stringify({free: Sys.free_ram(), total: Sys.total_ram()});
  print(topic, '->', msg);
  MQTT.pub(topic, msg, 1);
}, null);
```



Copy fs/init.js to the device and reboot the device:

```
mos put fs/init.js
mos call Sys.Reboot
```

In the AWS IoT console, click on "Test". Into the "Subscription topic" field, enter metrics/# and click "Subscribe to topic". See how messages are coming.

Direct device control via AWS IoT

If you're using Bash shell, you can use this alias to talk to your device interactively via AWS IoT:

```
alias mos1="mos --cert-file $(mos config-get mqtt.ssl_cert) --key-file $(mos config-get mqtt.ssl_key) --port mqtts://$(mos
mos1 ls -l
init.js 330
index.html 250
..."
```

For more in-depth explanation, see

[Secure remote device management with Mongoose OS](#)
blog post.

OTA update via AWS IoT device shadow

See [OTA via shadow](#)

Using AWS IoT in your custom firmware

The AWS IoT integration described above is implemented by the [aws](#) Mongoose OS library.

It provides device shadow API in both C/C++ and JavaScript, allowing developers to quickly prototype the device logic in JS before jumping to the C/C++ implementation.

Also, MQTT API is available, also for both C/C++ and JavaScript.

In order to have AWS IoT functionality in your custom application, make sure to add `aws` library to your `mos.yml` file:

```
libs:  
  ...  
  - origin: https://github.com/mongoose-os-libs/aws # <-- Add this!
```

Then, rebuild your app:

```
mos build
```

- [Amazon AWS IoT](#)
- [Google IoT Core](#)
- [IBM Watson](#)
- [Microsoft Azure IoT](#)
- [Private MQTT server](#)
- [Samsung Artik](#)

List all RPC services exported by device

```
mos call RPC.List  
[  
  "FS.Unmount",  
  ...
```

You can see what arguments an RPC service expects by calling `RPC.Describe`, which provides a simple introspection:

```
mos call RPC.Describe '{"name": "Config.Set"}'  
{  
  "name": "Config.Set",  
  "args_fmt": "{config: %M}"  
}
```

It works because C API for registering an RPC service has JSON format specification for the input arguments. That's why Mongoose OS knows about JSON formats for all registered RPC services.

Prepare Google Spreadsheet

- Create and empty Google spreadsheet
- Click on Tools / Script Editor, paste the following code:

```
function doPost(e) {  
  var doc = SpreadsheetApp.getActiveSpreadsheet();  
  var sheet = doc.getSheets()[0];  
  var row = JSON.parse(e.postData.contents) || []; // Parse POST data  
  row.unshift(new Date()); // Add timestamp as a first field  
  var r = sheet.getRange(sheet.getLastRow() + 1, 1, 1, row.length);  
  r.setValues([row]);  
  return ContentService.createTextOutput('');  
}
```

- Click "save". Choose a name if prompted
- In the menu, choose Publish / Deploy as web app.
Choose Project version: "New", Execute the app as: "Me", Who has

access: "Everyone, even anonymous". Click deploy

- On Authorization prompt, click "Review permissions", allow this application to see, edit, create and delete spreadsheets
- Click deploy. Copy the script URL, it should be in form <https://script.google.com/macros/s/xxxxxx/exec>

Add reporting logic to the firmware code

A firmware must send a POST request do the script URL.

A POST data must be a JSON array. Each element of the array will be inserted into a spreadsheet in a separate column. For example, if a POST data is [1234, "hello"], then in a spreadsheet a new row will be inserted with 3 columns: timestamp, 1234, and "hello".

This is a minimal app that reports a single value (free RAM) to the spreadsheet:

```
#include "mgos.h"

const char *s_url = "https://script.google.com/macros/s/xxxxxx/exec";

static void timer_cb(void *arg) {
    char buf[100];
    struct json_out out = JSON_OUT_BUF(buf, sizeof(buf));
    json_printf(&out, "[%d]", mgos_get_free_heap_size());
    mg_connect_http(mgos_get_mgr(), NULL, NULL, s_url, NULL, buf);
    (void) arg;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_set_timer(3000, MGOS_TIMER_REPEAT, timer_cb, NULL);
    return MGOS_APP_INIT_SUCCESS;
}
```

File system

Mongoose OS implements Virtual File System layer, VFS. That means it can attach (mount) different storage types into a single file system tree.

For example, a device can have an SPI flash storage and an SD card storage.

For each storage type, a filesystem driver must be implemented. For example, it is possible to write a driver that implements a Dropbox or Google Drive storage type, and a device (e.g. ESP8266 module) can mount a Dropbox folder.

Mongoose OS provides a

[Filesystem RPC service](#)

that allows remote filesystem management - for example, you can edit files remotely.

Adding file system

Let's see how to attach an additional filesystem to Mongoose OS in practice.

ESP8266

On ESP8266, Mongoose OS uses approximately 3MB of flash and default file system is 256K.

Most modules (ESP-12F have 4MB), so the last 1MB or so is available. Let's create a 512KB SPIFFS file system in this free space.

Note: we cannot use 1MB because at the end of flash there is system params area 16K in size which is used by SDK and SPIFFS requires size to be a power of 2.

First, flash the default firmware and verify that you have at least 4M of SPI flash, watch for the boot message:

```
$ mos flash esp8266 && mos console
Fetching https://mongoose-os.com/downloads/esp8266.zip...
...
[Jul 28 13:51:11.662] esp_mgos_init2      default 1.0 (20170728-103845/??)
[Jul 28 13:51:11.668] esp_mgos_init2      Mongoose OS 2017072810 (20170728-103845/??)
[Jul 28 13:51:11.675] esp_mgos_init2      SDK 2.1.0(ce90efd); flash: 4M; RAM: 52184 total, 49140 free
...
```

flash: 4M means we're good to go. There are modules with even bigger flash chips - [WEMOS D1 mini Pro](#) has 16MB, you can have extra 8MB filesystem there.

Now, create the file system:

```
$ mos call FS.Mkfs '{"dev_type": "sysflash", "fs_type": "SPIFFS", "fs_opts": "{\"addr\": 3145728, \"size\": 524288}"}
Using port /dev/ttyUSB0
null
```

addr is the offset from the beginning of the flash chip where to create the filesystem, 3145728 is just after the first 3M.

Mount the newly-created filesystem:

```
$ mos call FS.Mount '{"dev_type": "sysflash", "fs_type": "SPIFFS", "fs_opts": "{\"addr\": 3145728, \"size\": 524288}", "path": "/mnt"}'
Using port /dev/ttyUSB0
null
```

List files on the new FS - it will be empty:

```
$ mos ls -l /mnt
Using port /dev/ttyUSB0
```

Let's put a file there

```
$ mos put README.md
$ mos ls -l /mnt
Using port /dev/ttyUSB0
README.md 991
```

To make the device attach the file system automatically at boot, let's configure the `sys.mount` section:

```
$ mos config-set sys.mount.path=/mnt sys.mount.dev_type=sysflash sys.mount.fs_type=SPIFFS 'sys.mount.fs_opts={"addr": 3145728, "size": 524288}' &
Using port /dev/ttyUSB0
Getting configuration...
Setting new configuration...
Saving and rebooting...
Using port /dev/ttyUSB0
...
[Jul 28 14:02:32.787] esp_mgos_init2      default 1.0 (20170728-103845/??)
[Jul 28 14:02:32.793] esp_mgos_init2      Mongoose OS 2017072810 (20170728-103845/??)
[Jul 28 14:02:32.801] esp_mgos_init2      SDK 2.1.0(ce90efd); flash: 4M; RAM: 52184 total, 49140 free
[Jul 28 14:02:32.805] esp_print_reset_info Reset cause: 4 (soft reset)
[Jul 28 14:02:32.810] mgos_vfs_dev_open   sysflash () -> 0x3fff0034
[Jul 28 14:02:32.820] mgos_vfs_mount     Mount SPIFFS @ / (dev 0x3fff0034, opts {"addr": 32768, "size": 262144}) -> 0x3fff0044
[Jul 28 14:02:32.883] mgos_vfs_mount     /: size 233681, used: 99647, free: 134034
[Jul 28 14:02:32.970] mgos_sys_config_init MAC: 1AFE34A5930F
[Jul 28 14:02:32.975] mgos_sys_config_init WDT: 30 seconds
[Jul 28 14:02:32.979] mgos_vfs_dev_open   sysflash () -> 0x3fff0087c
[Jul 28 14:02:32.989] mgos_vfs_mount     Mount SPIFFS @ /mnt (dev 0x3fff0087c, opts {"addr": 3145728, "size": 524288}) -> 0x3fff0088c
[Jul 28 14:02:33.120] mgos_vfs_mount     /mnt: size 474641, used: 1255, free: 473386
[Jul 28 14:02:33.125] mgos_mdns_init    Listening on udp://:5353
...
```

We can make built-in HTTP server serve from /mnt instead of /:

```
$ mos config-set http.document_root=/mnt
```

ESP32

Typical ESP32 modules have 4M flash. Code size is bigger, there is approximately 400K available after all the system stuff. Let's add a 256K file system.

ESP32 uses [partition tables](#) for allocating flash space, we will need to add a partition and recompile firmware.

Check out or download the [default app](#), edit the `mos.yml` file and edit the esp32-specific section:

```
- when: mos.platform == "esp32"
  apply:
    build_vars:
      ESP_IDF_EXTRA_PARTITION: fs_ext,data,spiffs,,256K      # two lines      # Add these
```

Build and flash the firmware:

```
$ mos build --verbose --arch esp32 --clean && mos flash && mos console
...
Connecting to https://mongoose.cloud, user test
Uploading sources (2878 bytes)
...
Success, built default/esp32 version 1.0 (20170728-131414/???).
Firmware saved to build/fw.zip
...
[Jul 28 14:14:44.990] I (99) boot: Partition Table:
[Jul 28 14:14:44.990] I (110) boot: ## Label      Usage      Type ST Offset Length Flags
[Jul 28 14:14:44.990] I (135) boot: 0 nvs      WiFi data   01 02 00009000 00004000 00000000
[Jul 28 14:14:45.012] I (161) boot: 1 otadata   OTA data    01 00 0000d000 00002000 00000000
[Jul 28 14:14:45.012] I (186) boot: 2 app_0     OTA app     00 10 00010000 00180000 00000000
[Jul 28 14:14:45.013] I (212) boot: 3 fs_0     SPIFFS     01 82 00190000 00040000 00000000
[Jul 28 14:14:45.034] I (238) boot: 4 app_1     OTA app     00 11 001d0000 00180000 00000000
[Jul 28 14:14:45.035] I (263) boot: 5 fs_1     SPIFFS     01 82 00350000 00040000 00000000
[Jul 28 14:14:45.052] I (289) boot: 6 fs_ext    SPIFFS     01 82 00390000 00040000 00000000
[Jul 28 14:14:45.053] I (315) boot: End of partition table
...
```

Note presence of the extra `fs_ext` partition at the end.

Create the file system:

```
$ mos call FS.Mkfs '{"dev_type": "esp32part", "dev_opts": "{\"label\": \"fs_ext\"}", "fs_type": "SPIFFS"}'
Using port /dev/ttyUSB0
null
```

There is no need to specify size and offset, they are taken from partition information.

Mount the file system:

```
$ mos call FS.Mount '{"dev_type": "esp32part", "dev_opts": "{\"label\": \"fs_ext\"}", "fs_type": "SPIFFS", "path": "/mnt"}'
Using port /dev/ttyUSB0
null
```

To make the device attach the file system automatically at boot, let's configure the `sys.mount` section:

```
$ mos config-set sys.mount.path=/mnt sys.mount.dev_type=esp32part sys.mount.fs_type=SPIFFS 'sys.mount.dev_opts={"label": "f'...
```

There is no space available on the CC3200 chip for extra file system, but external SPI flash can be attached - see below.

External SPI flash

Mongoose OS has an [external SPI flash driver](#), which can be used to attach.

Having attached an SPI flash chip, you will be able to use the `spi_flash` driver like so:

```
$ mos call FS.Mkfs '{"dev_type": "spi_flash", "dev_opts": "{\"freq\": 20000000, \"cs\": 0}", "fs_type": "SPIFFS", "fs_opts": "{\"size\": 1048576}"}'
```

- [List RPC services](#)
- [Get device info](#)
- [Add filesystem](#)
- [MakerFactory ESP32 board](#)
- [Add FS with content](#)
- [Send data to spreadsheet](#)

HowTo: Create an image file of an populated filesystem

When developing with **Mongoose OS** and IoT devices to flash, it is often necessary to provide data that takes up more space than the average file system can provide. There exists a nice [HowTo](#) that explains how to add an additional filesystem to a device running **Mongoose-OS** - but the problem is: it's a lot of manual effort and it's a bit complicated. To make matters worse, the file system must also be "filled" manually for each individual device. So if you plan to flash many devices, you need to find a way to create a firmware package that already contains the file system. This will then be transferred to the device in one go - there is no need to intervene manually - everything is included in one firmware.

This HowTo describes how to achieve this goal. This requires some preparatory work and a few steps, but the result is worthwhile. And it doesn't matter if it's an additional 256kB file system or one with 16MB - in both cases you can cover it with a single firmware package and save a lot of trouble.

1. Necessary Preparations

The first step is to go the usual way - set up your device the way you want it, create the additional file system and store the desired files in it. There are two ways to do this:

- on the one hand the **conventional way** to create such a file system as described in [this manual](#), or
- on the other hand the **use of the library "fstab"**, which does the work of creating and mounting the file system for you.

It is sufficient to use the first way, because you only have to set up and mount the file system once. All in all, the effort is even a little less than with the external library, so here we describe the first way. A tutorial on how to use the "fstab" library may be created elsewhere, then you can decide yourself which way you want to go.

2. Creating and Populating an Additional Filesystem

This part is not very tiring: proceed as described under "[Adding File System](#)", set up the filesystem and mount it under the desired name, e.g. "**/mnt**" (this is our example) or another name as you like. The spelling of the filesystem reminds of a directory and is also used in the path syntax, but it is ultimately a mounting point for the files, the filesystem below is flat.

Once your device is newly flashed and the file system is active, you can test it with e.g.

```
mos[.exe] --port <portname> ls /mnt
```

If it is not available, an **error message** is displayed, if it is active, nothing is shown because it is empty - but without an error message.

Very simple.

Now you have to copy the desired files into the directory - there are a few points to consider. Files will be transferred to the device with the command

```
mos[.exe] --port <portname> put <local relative path>/<filename> <file system of the device>/<filename>
```

Unfortunately `mos[.exe]` may not transfer a whole bunch of files from a folder - every file has to be copied one after another with the full command line. So if you have to transfer a lot of files, it might be helpful to write this command down to a *shell script* or *batch file*, so you may edit and adapt the commands to your need and then execute them via the *shell script/batch file* - this will be much faster than doing it by hand.

Be aware: the command to put a file to an additional file system needs the complete target filename including the mountpoint. This is mandatory If you only use the mount point - so to speak as a target folder - no error will be shown, but the data will be copied into a file with an empty filename. This is kind of a bug, but you may avoid it.

So if you have finished this step successfully, you may examine your files system again with this command:

```
mos[.exe] --port <portname> ls -l /mnt
```

and you should see something like this (of course with different port and file names, but the example is clear):

```
macuser@Mac-Pro:~$ mos --port /dev/cu.SLABS_USB-to-UART ls -l /mnt
Bulb-green.dif 3024
Bulb-off.dif 3024
Bulb-red.dif 3024
Bulb-yellow.dif 3024
Switch-off.dif 11108
Switch-on.dif 11108
Titles-Ident.dif 6046
Titles.dif 18188
config.min.css.gz 1488
config.min.js.gz 1819
index.html 4962
logo.svg.gz 1142
mainswitch-status.json 19
tz-archive.zip 5220
workplaces.min.json 417
zepto.min.js.gz 10295
macuser@Mac-Pro:~$
```

The parameter `-l` tells `mos[.exe]` to list the file sizes too - it's a helpful information. With this process you have now created the "original file system", which is now used as the basis for the image file. For this reason you should be aware that all files are available in the desired version and form (e.g. gzipped).

3. Getting the Image

Now it's time to transfer the filesystem in the device to the local images file. So there has to be some information gathering first. When you boot your device and you have the serial monitor aka console active with this command:

```
mos[.exe] --port <portname> console
```

then you should see something like this scrolling in the terminals window:

The **red marked** entries are describing the filesystem you want to create an image from:

- "Name/Label" is the name or label of the existing filesystem, often also called "fs_ext"
- "Offset (HEX)" is the start address of the file systems data
- "Size (HEX)" is the size or length of the whole file system (even if it is empty)

With these parameters you may create the command to get the data to your local hard drive. For this you need the Python tool "esptool.py" from espressif. Since this tool is also used by mos[.exe], you will also have it on your system, but you can as well download it [from here](#). You'll find some documentation how to use the Python script too at this linked place. That Pathon must be installed is obvious, of course - this will not be further explained here.

```
esptool.py -p <portname> -b 1500000 read_flash 0x390000 0xC00000 mnt12mb.img
```

As you see, the parameters after read_flash are describing the start and the length of the data block to be read from the device and written to the file - these are the values we investigated before (see illustration above). In this example you see that a big file will be created - it is the additional file system of an ODROID-GO which comes with 16MB flash - as the system uses 4MB, there are still 12MB remaining for additional file system use. But even if you have an system like e.g. an ESP32 developer bord with 4MB, you may use a space of 256kB as external file system. This already may help a lot to put some files aside which e.g. you may use for web server purposes or symbols to be displayed on an TFT or OLED display. You may put the created image file e.g. in the srcsubfolder of your application folder, it will be used later.

4. Configuring mos.yml

Now we have to look for some settings in mos.yml to integrate the newly created image file into the build package fw.zip.

In this file you need to have the following lines in the build_vars section:

```
build_vars:  
ESP_IDF_EXTRA_PARTITION: data,data,spiffs,,12288K  
APP_EXTRA_FW_PARTS: fs_data:type=fs,src=src/mnt12mb.img,ptn=data
```

The first line should already exist as you have created the file system according to the mentioned HowTo. Perhaps the name may differ and you used fs_ext instead of data - but this won't play a big role. As you see, you have to enter the path of the created image file under the srcentry, and the name of the file system under ptn. As mentioned, you may exchange data found here with fs_ext in your own version. But be aware that the second parameter of the entry ESP_IDF_EXTRA_PARTITION has allways to be data no matter what the name is you've chosen for your file system.

After these settings you have to tell the system to mount the file system whwn booting - so you have to enter the following lines (if using another platform than ESP32 as in this example you have to look which parameters are valid in the particular case), for this use, you'll have to inlcude the library fstab:

```
config_schema:  
- ["fstab.fs0.dev", "data"]  
- ["fstab.fs0.type", "LFS"]  
- ["fstab.fs0.opts", '{"bs": 4096}']  
- ["fstab.fs0.path", "/mnt"]  
- ["fstab.fs0.create", false]
```

Under fstab.fs0.type you have to enter which file system type is used - this depends on the cretaed file systems in the first place. So in this example the file system type LFS (Little File System) is chosen for a bigger partition because it's way faster than SPIFFS and handles big file systems quite well. But as it wastes a lot of space, you should use SPIFFSfor smaller filesystems. The entry fstab.fs0.create has to be set to false , if not, the system will create a new on on start - so your existing data won't be used in the file system, because the new file system will be empty.

If you want to use LFS , you have to include the appropriate library in the configuration file:

```
libs:  
- origin: https://github.com/mongoose-os-libs/vfs-fs-lfs
```

Optionally - if you use the http server - you may enter also the following line unter config_schema:

```
- ["http.document_root", "/mnt"]
```

This will use the mounted file system as DOCROOT for the web server as well.

5. Goal reached!

So this is it! With these steps taken, after the next flash build process the image file will be integrated into the `fw.zip` package and flashed to the device. No further action necessary. So using this method enables you to create firmware packages with additional file systems which might be installed also via OTA without the need of individually handling the setup up of the file system.

6. Some Remarks and Hints

This example is based on the challenge to fully use the flash memory of an [ODROID-GO](#), a kind of "GameBoy like" designed device (ESP32, 16MB Flash, 4MB PSRAM).

So why not just define the standard filesystem bigger just like shown below if you have 16MB available?

```
build_vars:  
ESP_IDF_SDKCONFIG_OPTS: >  
${build_vars.ESP_IDF_SDKCONFIG_OPTS}  
CONFIG_ESPTOOLPY_FLASHSIZE_4MB=  
CONFIG_ESPTOOLPY_FLASHSIZE_16MB=y  
CONFIG_ESPTOOLPY_FLASHSIZE="16MB"  
FLASH_SIZE: 16M  
FS_SIZE: 6291456
```

The explanation: Simply setting the value of `FLASH_SIZE` and `FS_SIZE` to higher values using all the space was not just like this possible, because it will lead to a division of the flash (because of the OTA architecture, I assume) and you can only use half of it.

Anything over **6MB** (half the space of the remaining **12MB** available after subtraction of **4MB** needed by the system from the **16MB** total) led to build process errors. In addition, the standard file system is formatted with `SPIFFS`, which leads to long processing times for large partitions, which sometimes even cause the watchdog to be activated. For this reason, it is recommended to use the file system `LFS` ("Little File System"), but this must then be set up as an additional file system.

And even if you have only a little amount of extra space, it will be better to add this as an additional file system, because if you increase the standard file system, you only may use half of the remaining space, for the reasons explained above.

7. Troubleshooting

If you have already stored a file system in the older LFS version 1, this won't work out of the box like described above. In this case you may set

```
cdefs:  
MGOS_LFS1_COMPAT: 1
```

in the `mos.yml` file, so the driver will be compatible with the old version. But this is only a solution to perhaps convert old data, in daily use, the newer version is the better choice.

MakerFactory board + Mongoose OS

This tutorial demonstrates how to use Mongoose OS on MakerFactory ESP32 development board, and use Google IoT core as a cloud backed to control the board and report metrics.

Setup the board and Google IoT Core

Please follow [Google IoT Core tutorial](#)

in order to setup the cloud side and provision your board to Google IoT Core.

You need to complete **Setup Google IoT Core** and

Setup device sections only. When done, continue on this tutorial.



NOTE: before executing `mos flash` command, press and hold IO0 button on the board in order to enter programming mode. That needs to be done every time you flash the board.

HINT FOR THE MANUFACTURER:

in order to omit manual button press, modify USB-to-Serial adapter wiring by connecting

DTR to GPIO0 and RTS to RESET. See notes from Espressif at

<https://github.com/espressif/esptool/wiki/ESP8266-Boot-Mode-Selection#automatic-bootloader>

Controlling LED via Google IoT Core

Open `fs/init.js` in your favorite editor, copy-paste the following snippet and save:

```
load('api_config.js');
load('api_gpio.js');
load('api_mqtt.js');
load('api_neopixel.js');

let strip = NeoPixel.create(33, 2, NeoPixel.GRB);
strip.clear();

let topic = '/devices/' + Cfg.get('device.id') + '/config';
MQTT.sub(topic, function(conn, topic, msg) {
  print('Topic:', topic, 'message:', msg);
  // Expected config format: {"led": 0, "r": 123, "g": 123, "b": 123}
  let obj = JSON.parse(msg) || {};
  strip.setPixel(0, obj.r || 0, obj.g || 0, obj.b || 0);
  strip.show();
}, null);
```

The snippet above initialises RGB LED strip, and subscribes to the `/config` object notifications. Once the config object is sent to device by Google IoT Core, the handler function is called and lights the corresponding LED.

In the command prompt (or terminal on Linux/Mac), enter the following commands

to copy `init.js` to the device, reboot the device, and start monitoring

serial output:

```
mos put fs/init.js
mos call Sys.Reboot
mos console
```



Find a registered device in a Google IoT Core device registry.
Click on "Update Config" button, and in the opened dialog, enter
the following configuration and hit "SEND TO DEVICE":

```
{"r": 50, "g": 0, "b": 0}
```

Note how one of the LEDs turns red. Enter

```
{"r": 0, "g": 0, "b": 0}
```

Note how the LED turns off.

Important thing to note: Google IoT Core send device config object each time device comes online or a config change is made. That means, that if you power cycle the device, it'll sync its LED state with the cloud as soon as it gets connected.

Reporting state to Google IoT Core

We are going to connect a Grove DHT temperature sensor to the board, then modify `fs/init.js` to periodically read the temperature and send it over to the Google IoT Core.

Get the Grove DHT sensor and connect it to the IO17 Grove connector on the board, as shown on the picture. Make sure to enable IO17 via the switch, as shown on the picture.

The DHT driver is built-in to the `demo-js` app we're using, therefore we do not need to rebuild the firmware.

Open `fs/init.js` in your favorite editor, copy-paste the following snippet and save:

```
load('api_config.js');
load('api_dht.js');
load('api_mqtt.js');
load('api_timer.js');

let topic = '/devices/' + Cfg.get('device.id') + '/state';
let pin = 17; // GPIO pin which has a DHT sensor data wire connected
let dht = DHT.create(pin, DHT.DHT11); // Initialize DHT library

Timer.set(3000 /* milliseconds */, Timer.REPEAT, function() {
  let t = dht.getTemp();
  let h = dht.getHumidity();
  if (isNaN(h) || isNaN(t)) {
    print('Failed to read data from sensor');
  } else {
    let msg = JSON.stringify({temperature: t, humidity: h});
    print(topic, '->', msg);
    MQTT.pub(topic, msg, 1);
  }
}, null);
```

to copy `init.js` to the device, reboot the device, and start monitoring serial output:

```
mos put fs/init.js
mos call Sys.Reboot
mos console
```

In the Google IoT Web console, click on the "Configuration and state history", and see how new state objects are arriving. Click on any state object, select "text" view, and check the JSON string that is sent by device.

Get device information

The solution is to call `Sys.GetInfo` RPC service.

From your terminal, run:

```
mos call Sys.GetInfo
[Aug 21 08:29:17.016] {
...
[Aug 21 08:29:17.016]   "mac": "5ECF7F060046",
[Aug 21 08:29:17.016]   "arch": "esp8266",
[Aug 21 08:29:17.016]   "wifi": {
[Aug 21 08:29:17.016]     "sta_ip": "192.168.1.31",
...
}
```

You can see e.g. MAC and IP addresses in the output.

In order to call the same function in JS, do:

```
load('api_rpc.js');
RPC.call(RPC.LOCAL, 'Sys.GetInfo', null, function(resp, ud) {
  print('Response:', JSON.stringify(resp));
  print('MAC address:', resp.mac);
}, null);
```

The device would print to the output:

```
[Aug 21 08:23:33.162] Response: { ... "arch":"esp8266",... }
[Aug 21 08:23:33.170] MAC address: 5ECF7F060046
```

Advanced guide: C/C++

In this guide we will learn how to implement device logic in C/C++.

In particular, our firmware will:

- have an ability to read DHT22 temperature/humidity sensor data
- set up a timer to read and print sensor data to the debug console
- implement a RPC service for reading the sensor remotely - for example, via MQTT

[1. Start mos tool](#)

[2. Clone template project](#)

[3. Learn app dir structure](#)

[4. Add required libraries](#)

[5. Create app's custom config](#)

[6. Add sensor code](#)

[7. Build app, test sensor](#)

8. Add RPC service

9. Rebuild app and test RPC

Start mos tool

Run `mos` tool without arguments to start the Web UI. After start, the Web UI changes current working directory to the directory where it finishes the last time. In other words, it "remembers" its settings: the working directory, chosen port, board, etc.

In this example, it is an `app1` directory, a quickstart example I have done recently:

Since we are going to create our new app in a different directory, use the `cd DIRECTORY` command to change the current directory.

I am going to do `cd ..` to go up one level. Notice the current directory change:

Clone template project

Now we are going to use a `mos clone URL DIRECTORY` command, in order to clone some remote app into a `DIRECTORY`. Press `Ctrl-n` - that populates the input field with `mos clone https://github.com/mongoose-os-apps/demo-js app1`. We don't want to use `demo-js` as a template, so change it to `empty` to use a minimal app, and change `app1` to `app2`:

Now press Enter to execute the command. Notice that the `mos` tool automatically changes enters into the cloned directory:

Learn app directory structure

Click on the folder icon on the bottom left corner to open a system file browser in the current directory:

Here is the meaning of all files:

```
fs/      -- All files we put here, will end up on device's filesystem
└ index.html -- Device's HTTP server, if enabled, will serve this file
LICENSE
mos.yml    -- Describes how to build an app
README.md   -- Document your app in this file
src/
└ main.c    -- Contains device logic. We are going to edit this file
```

Add necessary libraries

Open `mos.yml` file in your favorite editor and add support for DHT sensor:

```
libs:
- origin: https://github.com/mongoose-os-libs/rpc-service-config
- origin: https://github.com/mongoose-os-libs/rpc-service-fs
- origin: https://github.com/mongoose-os-libs/rpc-uart
- origin: https://github.com/mongoose-os-libs/wifi
- origin: https://github.com/mongoose-os-libs/dht # <-- Add this line!
```

Note - all available libraries are collected under the <https://github.com/mongoose-os-libs> organisation. They are categorised

and documented under the "API Reference" docs section.

Now, open `src/main.c`, you'll see the following

skeleton code which initialises an app that does nothing:

```
#include "mgos.h"

enum mgos_app_init_result mgos_app_init(void) {
    return MGOS_APP_INIT_SUCCESS;
}
```

Create custom configuration section

Let's add a code that reads from a DHT temperature sensor every second.

The pin to which a sensor is attached, we make configurable by editing

a `config_schema:` section in the `mos.yml`, to have it like this:

```
config_schema:
- ["app.pin", "i", 5, {title: "GPIO pin a sensor is attached to"}]
```

This custom configuration section will allow us to change sensor pin at run time, without recompiling firmware. That could be done programmatically or via the `mos` tool, e.g. `mos config-set app.pin=42`.

NOTE: see [mos.yml file format reference](#)
for the full documentation about the `mos.yml`.

Add code that uses sensor library

Then, edit `src/main.c`, add a timer (see [timer api docs](#)) that reads DHT and logs the value (error handling is intentionally omitted):

```
#include "mgos.h"
#include "mgos_dht.h"

static void timer_cb(void *dht) {
    LOG(LL_INFO, ("Temperature: %lf", mgos_dht_get_temp(dht)));
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_dht *dht = mgos_dht_create(mgos_sys_config_get_app_pin(), DHT22);
    mgos_set_timer(1000, true, timer_cb, dht);
    return MGOS_APP_INIT_SUCCESS;
}
```

The `mgos_dht.h` file comes from the `dht` library that we have included to our app.

In order to find out its documentation and API, navigate to

"API Reference" -> "Drivers" -> "DHT temp sensor". This should bring you to

[this page - DHT temp sensor](#). Similarly, you can find

out about any other library.

Build the app and test sensor reading

Connect DHT sensor to pin 5. The sensor itself has following pins:

This is an example with ESP8266 NodeMCU. Red connector is VCC 3.3 volts, black connector is ground GND, and yellow is data, connected to pin 5:

Build, flash the firmware, and attach the console to see device logs. Assume we're working with ESP8266:

Choose your board and port in the the UI, and run `mos build` command:

When finished, run `mos flash` to flash the firmware and see the output in the console:

Add RPC service to read sensor data remotely

Now let's use the cornerstone of Mongoose OS remote management capabilities.

We can make any hardware function be remotely accessible. This is done by creating an RPC service. Read more about it in the Overview and Core libraries sections, and here we jump straight to it. Looking at

[MG-RPC API doc](#), add RPC service `Temp.Read`:

```
#include "mgos.h"
#include "mgos_dht.h"
#include "mgos_rpc.h"

static void timer_cb(void *dht) {
    LOG(LL_INFO, ("Temperature: %lf", mgos_dht_get_temp(dht)));
}

static void rpc_cb(struct mg_rpc_request_info *ri, void *cb_arg,
                   struct mg_rpc_frame_info *fi, struct mg_str args) {
    mg_rpc_send_responsef(ri, "{value: %lf}", mgos_dht_get_temp(cb_arg));
    (void) fi;
    (void) args;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_dht *dht = mgos_dht_create(mgos_sys_config_get_app_pin(), DHT22);
    mgos_set_timer(1000, true, timer_cb, dht);
    mg_rpc_add_handler(mgos_rpc_get_global(), "Temp.Read", "", rpc_cb, dht);
    return MGOS_APP_INIT_SUCCESS;
}
```

Build and test the final app

Run `mos build` followed by `mos flash`.

And now, call the device's RPC service by running `mos call Temp.Read`.

You will see `{"value": 18.6}` printed.

This call could be performed over the serial connection as well as over network connection - see [RPC section](#) to learn more.

Supported hardware

Mongoose OS supports the following microcontrollers:

Vendor	Microcontrollers
STMicroelectronics	STM32L4, STM32F4, STM32F7 series
Texas Instruments	CC3220, CC3200
Espressif Systems	ESP32, ESP8266

Development boards reference

STM32 B-L475E-IOT01A

Capabilities	WiFi, BT, 868 RF, NFC, 128k RAM, 1M Flash
Flash demo-js app	<code>mos flash stm32-B-L475E-IOT01A</code>
Build an app	<code>mos build --platform stm32 --board B-L475E-IOT01A</code>

[Buy on mouser.com](#)

STM32 NUCLEO-F746ZG

Capabilities	Ethernet, 320k RAM, 1M Flash
Flash demo-js app	<code>mos flash stm32</code>
Build an app	<code>mos build --platform stm32 --board NUCLEO-F746ZG</code>

[Buy on mouser.com](#)

TI CC3220SF LaunchPad

Capabilities	WiFi, 256k RAM, 1M Flash
Flash demo-js app	<code>mos flash cc3220</code>
Build an app	<code>mos build --platform cc3220</code>

[Buy on mouser.com](#)

ESP32 PICO-D4-KIT

Capabilities	WiFi, BT, 520k RAM, 4M Flash
Flash demo-js app	<code>mos flash esp32</code>
Build custom app	<code>mos build --platform esp32</code>

[Buy on mouser.com](#)

ESP8266 NodeMCU

Capabilities	WiFi, 96k RAM, 4M Flash
Flash demo-js app	<code>mos flash esp8266</code>

Build an app

```
mos build --platform esp8266
```

[Buy on aliexpress.com](#)

ESP32 Devkit-C

Capabilities	WiFi, BT, 520k RAM, 4M Flash
Flash demo-js app	<code>mos flash esp32</code>
Build an app	<code>mos build --platform esp32</code>

[Buy on mouser.com](#)

ESP32 ODROID-GO

Capabilities	WiFi, BT, 2.4" TFT, 4MB PSRAM, 16M Flash, SD card, Battery, Speaker, 10 Buttons, Powerswitch
Flash demo-js app	First, download conf3.json from here . Then, <code>mos flash esp32</code> <code>mos put conf3.json</code>
Build an app	<code>mos build --platform esp32</code>



[Buy on hardkernel.com](#)

Mongoose OS quick start guide

A 12-minute guide to turn your device into a mobile-controllable, updatable, remotely manageable, secure, configurable product.

1. Download and install [1 min]
2. Start mos tool [0 min]
3. Install drivers [2 min]
4. Create new app [1 min]
5. Build app firmware [1 min]
6. Flash firmware [1 min]
7. Configure WiFi [0 min]
8. Register on mDash [1 min]
9. Change firmware [2 min]
10. Update over-the-air [2 min]

1. Download and install mos tool

Mongoose OS uses `mos` tool for various tasks:

building firmware, flashing firmware,

managing device, provisioning device on cloud services, and so on.

OS	Setup instructions
	Create C:\mos folder. Right-click on this mos.exe link, choose "Save link as", save <code>mos.exe</code> into the C:\mos folder.

Windows	Double-click on <code>mos.exe</code> to start a Web UI. If it does not start, open command prompt, enter <code>cd c:\mos</code> and then <code>mos --start-webview=false</code>
MacOS	First, install brew utility . Then execute the following to start a Web UI: <code>brew tap cesanta/mos</code> <code>brew install mos</code> <code>mos</code>
Ubuntu Linux	Note: Linux version starts UI in a browser window. <code>sudo add-apt-repository ppa:mongoose-os/mos</code> <code>sudo apt-get update</code> <code>sudo apt-get install mos</code> <code>mos</code>
Arch Linux	Note: Linux version starts UI in a browser window. <code>git clone https://github.com/mongoose-os/mos</code> <code>cd mos-tool/mos/archlinux_pkgbuilder/mos-release</code> <code>makepkg</code> <code>pacman -U ./mos-*.tar.xz</code> <code>mos</code>
Generic MacOS/Linux	Note: starts UI in a browser window. <code>curl -fsSL https://mongoose-os.com/downloads/mos/install.sh /bin/bash</code> <code>mos</code>

2. Start mos tool

Make sure your device is connected via the USB to your workstation.

Completing previous section should leave you with the `mos` Web UI running:

Note: if you like using command prompt / terminal instead of the UI,
you can execute `mos` commands in a command prompt instead.

The Web UI appears only when `mos` tool is started without arguments.

On Windows and Mac, closing the Web UI window also terminates the `mos` command.

On Linux, the Web UI is started in a browser, so to stop it, close the browser window and pressing `Ctrl-C` in a terminal where `mos` is running.

3. USB-to-Serial drivers

If you have your device connected via the USB, the UI can still complain about the unavailable ports like this:

The most usual reason for this is a missing USB-to-Serial driver. Below are the links to the driver installation pages. Install the driver, and re-insert your device into the USB.

- [Silabs drivers](#) for Espressif boards
- [CH43x drivers](#) for Espressif boards
- [FTDI drivers](#) for CC3200, CC3220

In a "Choose port" dropdown, at least one port must appear.

NOTE: on some Mac computers, the port might not show up even after the driver is installed. The possible reason could be the extension policy, the fix is covered in [this issue](#)

4. Create new app

In the UI, select your board from the dropdown menu. Then, press `Ctrl-n` to create a new app. That will paste the following command:

```
mos clone https://github.com/mongoose-os-apps/demo-js app1
```

Press Enter. That is going to clone `demo-js` repo into the `app1` directory.

Notice how current directory changes to `app1`:

5. Build app firmware

Make sure the board is selected. Type `mos build` and press Enter.

That starts a build process that takes up to a minute to complete.

When the command finishes, the built firmware will be located at `build/fw.zip`.

As you can see, we did not make any changes to the app - we'll do that later.

6. Flash firmware

Make sure the port is selected in the dropdown menu.

Type `mos flash` and press enter. That will flash a built firmware to the device:

When flashing completes, the firmware starts to write logs to the serial console. Notice how new messages start to appear in the right window.

The window autoscrolls to the bottom, unless you scroll up manually to see some previous logs.

7. Configure WiFi

Type command `mos wifi WIFI_NETWORK_NAME WIFI_PASSWORD`. Use your own network name/password. Press enter. On the serial console, you'll see how the device reboots after saving new configuration settings, joins the WiFi network, gets the IP configuration, and synchronises time with SNTP server:

8. Add device to the mDash management dashboard

- Login to the mDash at <https://mdash.net/>.
- Click on "Add device" button. That'll create a new device.
- Click on a gears icon to trigger device management dialog.
- Click on a "Token" link to copy the device's access token to the clipboard.
- Go back to the mos tool, type command (change TOKEN to your copied token) and press enter:

```
mos config-set dash.enable=true dash.token=TOKEN
```

A device should become online ("green") on a dashboard:

Click on a gears icons to see a management dialog for a device:
 general information, configuration editor, file
 editor, device shadow editor, and an RPC service call window. We won't study
 all these management tools now though, so proceed to the next step.

9. Make changes to the firmware

Click on the folder icon on the bottom left corner. That opens a system
 file browser in the current app directory. Open `fs/init.js` in your
 favorite editor:

Paste this code snippet, that sends an MQTT message every second:

```
load('api_timer.js');
load('api_mqtt.js');

Timer.set(1000, Timer.REPEAT, function() {
    let ok = MQTT.pub('test/topic', 'hello from an updated firmware');
    print('mqtt message sent?', ok);
}, null);
```

Then, rebuild the firmware with `mos build`.

10. Update firmware over-the-air

Now let's update our device with the new firmware over the air.

Go to the mDash, click on the firmware update icon:

A file dialog should appear. In the file dialog, navigate to the
`app1/build` directory, and choose `fw.zip` file:

Notice how the progress bar appears showing the OTA progress:

During the OTA process, the device receives chunks of new firmware
 from the cloud, and serial console displays that:

Eventually, the device reboots with the new firmware and we could
 see the changed message in the serial console:

Next steps

It's time to move on to more advanced topics - learn how to add support

to different hardware peripherals, write logic in both C/C++ and JS, use API reference, send data to the databases/analytics engines like Firebase. Proceed to the [advanced guide](#).

Mongoose OS IDE

Mongoose OS provides an experimental Visual Studio Code plugin, that integrates Mongoose OS support into the VSCode.

To install it, run VSCode, search and install a "Mongoose OS IDE" extension:

Code completion

There is C/C++ and JavaScript completion support that makes writing Mongoose OS code a better experience.

The completions are made by scanning Mongoose OS core code, and all contributed libraries.

Example apps

GitHub repo	Description	Author
LilyGo-HiGrow-Sensor-v1	LilyGo HiGrow ESP32 Plant Sensor v1	Jason Harrell info@latitude17.io
alexa-microwave	Control a microwave with ESP32 via MQTT / Amazon Alexa	Steve Kasuya
aws-iot-button	Internet button on AWS IoT	mongoose-os
aws-iot-heater	Smart heater on AWS IoT	mongoose-os
aws-pico	ESP32-PICO-KIT breakout board demo for AWS IoT	Angus Gratton
aws-uart	Reading UART0 and sending the data to AWS IoT	bravokeyl bravokeyl@gmail.com
blynk	Using Blynk mobile app with Mongoose OS	mongoose-os
blynk-bme280-js	Using BME280 sensor with Blynk mobile app	Rumen Nikiforov
captive-portal	Captive portal example	mongoose-os
demo-c	A demo Mongoose OS firmware for C/C++	mongoose-os
demo-js	A JS-enabled demo Mongoose OS firmware	mongoose-os
door-bell	IoT doorbell that works via MQTT - generic MQTT server, AWS IoT, Google IoT, etc	Steve Kasuya steve.kasuya@gmail.com
door-sensor	IoT Door Sensor using a MC-38 (JavaScript)	Ben Rockwood benr@cuddletech.com
empty	A Mongoose OS app skeleton	mongoose-os
environmental-monitor	Environmental Monitor using a BME280 Sensor & SSD1306 OLED	Ben Rockwood benr@cuddletech.com
example-arduino-adafruit-ads1x15-js	Adafruit ADS1X15 usage example (JavaScript)	djzxz@wp.pl
example-arduino-adafruit-bme280-c	BME280 sensor usage example (C/C++)	mongoose-os

example-arduino-adafruit-bme280-js	BME280 sensor usage example (JavaScript)	mongoose-os
example-arduino-adafruit-ssd1306-c	Adafruit SSD1306 usage example (C/C++)	mongoose-os
example-arduino-adafruit-ssd1306-js	Adafruit SSD1306 usage example (JavaScript)	mongoose-os
example-arduino-dallas-temperature-c	Dallas Temperature sensor usage example (C/C++)	mongoose-os
example-arduino-dallas-temperature-js	DallasTemperature sensor usage example (JavaScript)	mongoose-os
example-arduino-ds3231-js	DS3231 usage example (JavaScript)	djzxz@wp.pl
example-arduino-hello-c	Arduino C/C++ basic API usage example	mongoose-os
example-arduino-onewire-c	Arduino OneWire C app	mongoose-os
example-arduino-onewire-js	Arduino OneWire API usage example (JavaScript)	mongoose-os
example-arduino-wire-c	Arduino Wire API usage example on Mongoose OS (C/C++)	mongoose-os
example-binary-lib-c	An example demonstrating usage of binary libs (.a)	mongoose-os
example-dht-c	DHT sensor usage example (C/C++)	mongoose-os
example-dht-js	DHT sensor usage example (JavaScript)	mongoose-os
example-dns-sd	DNS-SD usage example in C/C++	mongoose-os
example-m5stack-js	A JS-enabled demo application for M5stack	mongoose-os
example-neopixel-c	Neopixel strip control example	mongoose-os
example-neopixel-js	Neopixel strip control example	mongoose-os
example-no-libs-c	Most minimal app - no libs, no networking, no RPC, nothing	mongoose-os
example-rpc-c	Custom RPC service example in C/C++	mongoose-os
example-shadow-js	A device shadow example in JavaScript	mongoose-os
example-spi-c	SPI usage example in C/C++	mongoose-os
example-uart-c	UART usage example in C/C++	mongoose-os
example-uart-js	UART JavaScript API usage example	mongoose-os
example-wifi-scan-js	Wifi JavaScript API usage example	mongoose-os
http-fetch	Download URL and write content to the UART or file	mongoose-os
huzzah-featherwing	A Mongoose-OS Huzzah TFT Featherwing demo	Pim van Pelt pim@ipng.nl
lilygo-higrow-soil-temp-humi-sensor	HiGrow Plant Sensor	Luca Fabbri luca@higrow.tech
losant-motion-sensor	Losant - detecting Motion Using a PIR sensor	Taron Foxworth
losant-mqtt	Losant cloud service integration	Taron Foxworth
losant-temp-sensor	App to send ESP32/8266 temperature readings to Losant periodically.	Utkarsh Verma (utkarshverma@protonmail.com)
mq135	Using MQ135 gas sensor with Mongoose OS	Moncef AOUDIA (mf.aoudia@gmail.com)
neopixel-aws-iot	Control Neopixels from AWS IoT and an Android Companion App	anelson (austin@mgenio.com)

ota-aws-shadow	An empty app that does nothing	mongoose-os
smallest-wifi-button	The Smallest WiFi Button in the World (ESP8266, MQTT, IFTTT)	bitluni
smart-pool-pump	Smart Pool Timer Controller, integrates AWS shadow, BMP180 and ADE7912	Chizuruoke Chikwendu only1chi@gmail.com
smart-washing-machine	Making washing machine smart with WEMOS D1 mini	bitluni
smoke-detector	Smoke Detector (ESP8266, MQTT, IFTTT)	bitluni
sonoff-basic-openhab	Sonoff Basic firmware to work with openHAB	Michael Fung hkuser2001@gmail.com
wifi-scan	Scan WiFi networks on button press	mongoose-os
wifi-setup-web	Setup WiFi via Web UI	mongoose-os

- [Quick Start](#)
- [Develop in C/C++](#)
- [Develop in JavaScript](#)
- [IDE support](#)
- [Supported hardware](#)

Advanced Guide - JS

In this section we are going to implement the same functionality as in the previous section, but using JavaScript instead of C/C++. Also, we learn how to export custom C/C++ functions into JavaScript.

Please refer to the previous section on how to connect the DHT sensor.

Clone template project

To clone the template project, repeat the same steps as in the previous section:

```
git clone https://github.com/mongoose-os-apps/empty my-app
Cloning into 'my-app'...
remote: Counting objects: 68, done.
remote: Total 68 (delta 0), reused 0 (delta 0), pack-reused 68
Unpacking objects: 100% (68/68), done.
cd my-app
```

Add necessary libraries

In the previous step, we added DHT library. Do that, and also add mJS library to enable JavaScript support. Open `mos.yml` file in your favorite editor and add support for DHT sensor and JavaScript:

```
libs:
  - origin: https://github.com/mongoose-os-libs/rpc-service-config
  - origin: https://github.com/mongoose-os-libs/rpc-service-fs
  - origin: https://github.com/mongoose-os-libs/rpc-uart
  - origin: https://github.com/mongoose-os-libs/wifi
  - origin: https://github.com/mongoose-os-libs/dht # <-- Add this line!
  - origin: https://github.com/mongoose-os-libs/mjs # <-- Add this line!
```

Create custom configuration section

Same as in the previous section, add custom `app.pin` config setting by editing `mos.yml`:

```
config_schema:  
- ["app", "o", {title: "My app custom settings"}]  
- ["app.pin", "i", 5, {title: "GPIO pin a sensor is attached to"}]
```

Write device logic

Create `fs/init.js` file with the following content

(see [MG-RPC JS API reference](#)):

```
load('api_config.js');  
load('api_rpc.js');  
load('api_dht.js');  
load('api_timer.js');  
  
let pin = Cfg.get('app.pin');  
let dht = DHT.create(pin, DHT.DHT22);  
  
Timer.set(1000, true, function() {  
  print('Temperature:', dht.getTemp());  
}, null);  
  
RPC.addHandler('Temp.Read', function(args) {  
  return { value: dht.getTemp() };  
});
```

Tip: keep variable and function declaration names under 5 characters or less to minimise JavaScript memory usage.

mJS uses `double` type to hold values. Valid double hold numbers, and `NaN` values hold other types. 2 bytes out of 8 are used for type indication, and the rest

6 bytes are used for payload which depends on type.

Short strings that are 5 chars or less + terminating `\0` are stored directly in the `NaN` payload.

Longer strings are stored in a special buffer, and `NaN` payload stores a pointer to that buffer. Therefore, strings of length 5 and less take no extra memory in mJS.

Build and test the final app

Build and flash the app:

```
mos build --arch esp8266 && mos flash
```

And now, call the device's RPC service:

```
mos call Temp.Read  
{"value": 18.6}
```

Export custom function

[mJS JavaScript engine](#), which is used by Mongoose OS, allows to export C/C++ functions directly into JavaScript

via the FFI (foreign function interface) mechanism. Let's show how it is done.

Edit `src/main.c` file and add `my_func()` function there:

```
int my_func(int a, int b) {
    return a + b;
}
```

Note: if you want to export a C++ function, make sure it is declared as `extern "C"`. Add the following snippet in `fs/init.js`:

```
let f = ffi('int my_func(int, int)');
print('Calling C my_func:', f(1,2));
```

Rebuild, flash, test the app:

```
mos build --arch esp8266 && mos flash && mos console
[Aug  9 09:02:05.399] mgos_init           Init done, RAM: 52680 total, 44404 free, 44288 min free
[Aug  9 09:02:05.753] Calling C my_func: 3
```

How FFI calls are implemented

This is how it is done:

- During the build phase, Mongoose OS toolchain scans all JavaScript files present in the project.
- All `ffi(...)` JavaScript calls are collected in one list, and `build/gen/ffi_exports.c` file is generated with the signatures of C functions that JavaScript code intends to call.
- A `mgos_dlsym()` function is auto-generated that returns the address of the C function by its name, which makes `ffi()` call possible.

The `mos build` command has created `build` directory. Open `build/gen/ffi_exports.c` file and notice a line with `my_func`:

```
const struct mgos_ffi_export ffi_exports[] = {
...
{"my_func", my_func},
...
}
```

To summarise, this section demonstrates how to use JavaScript API, libraries, and how to export custom C functions.

- [Quick Start](#)
- [Cloud Tutorials](#)
- [User Guide](#)
- [API Reference](#)
- [HOWTOs](#)

Debugging tools

Mongoose OS provides several ways to debug firmware functionality. They are listed below in the order of complexity.

Increase debug log level

There is a top-level `debug` section in the configuration. It could be inspected by clicking on a "Device config" tab in the GUI, or through the command line:

```
mos config-get debug
{
    "factory_reset_gpio": -1,
    "level": 2,
    "mbedtls_level": 0,
    "mg_mgr_hexdump_file": "",
    "stderr_topic": "esp8266_067082/log",
    "stderr_uart": 0,
    "stdout_topic": "esp8266_067082/log",
    "stdout_uart": 0,
    "udp_log_addr": ""
}
```

The default level is 2. Level -1 means no log. Level 0 is `ERROR`, 1 is `WARN`, 2 is `INFO`, 3 is `DEBUG`, 4 is `VERBOSE_DEBUG`. Level 4 means log everything possible, it is very verbose. Start with increasing to level 3.

Increasing the debug level from the command line

```
mos config-set debug.level=3
Getting configuration...
Setting new configuration...
Saving and rebooting...
```

Debug configuration settings quick reference

```
{
    "factory_reset_gpio": -1,           // Used to reset the device. Hold this pin down and reboot to reset
    "filter": "mg_,*mqtt*",            // Log filter. Prefix match glob for source file or function name
    "level": 2,                      // Log verbosity level
    "mbedtls_level": 0,               // Log verbosity level for TLS library
    "mg_mgr_hexdump_file": "",        // Set to "-" to hexdump network packets
    "stderr_topic": "",                // If set, an MQTT topic to send stderr logs to
    "stderr_uart": 0,                  // UART number for stderr logs
    "stdout_topic": "",                // If set, an MQTT topic to send stdout logs to
    "stdout_uart": 0,                  // UART number for stdout logs
    "udp_log_addr": ""                // Set to "IP_ADDR:PORT" to send logs to this UDP address
}
```

Notes:

- `mos` tool starts a UDP listener on port 1993. Therefore, if you set `debug.udp_log_addr` to `YOUR_MACHINE_IP:1993`, `mos` tool will start showing your device logs.
- If you connect to your device using via Websocket, as `ws://IP_ADDR/rpc`, then `mos` tool automatically sets the `debug.udp_log_addr` to the correct value and you don't need to set it up manually.

Analysing core dumps

If your app crashes showing a core dump message, you can start GDB against that coredump and analyse the backtrace at the time of the crash, as well as the state of the firmware.

First, save the coredump data to the `console.log` file.

Then clone the `mongoose-os` repo:

```
git clone https://github.com/cesanta/mongoose-os
```

Then from your apps' directory, do

```
make -C PATH_TO_MONGOOSE_OS_REPO/fw/platforms/esp8266 debug_coredump \
CONSOLE_LOG=$(pwd)/console.log ELF_FILE=$(ls $PWD/build/objs/fw.elf)
```

Running a heap log analyser

Heap log analyser is a tool developed by the Mongoose OS team. It is used for detecting memory leaks, and does it in a visual way, allowing to quickly pinpoint the place where a leak is happening.

ESP8266 is the only device this is supported on.

In order to use the heap log viewer, follow the steps below:

Enable heap log tracing

Edit your app's `mos.yml`, add the following section:

```
build_vars:
  MGOS_ENABLE_HEAP_LOG: 1
  MGOS_ENABLE_CALL_TRACE: 1
```

Rebuild your app:

```
mos build --clean --arch esp8266
...
```

Run your app to collect the heap log

Your console log will become very noisy lots of `hl{...}` lines.

Run your app, let it run a few cycles, and save the log somewhere.

If you're on Mac/linux,

```
mos console | tee /tmp/console.log | grep -av 'hl{'
```

This way you don't see the noise, but it's all logged in `/tmp/console.log`.

Analyse logs

Clone `mongoose-os` repo and start heap log viewer tool:

```
git clone https://github.com/cesanta/mongoose-os
cd mongoose-os/tools/heaplog_viewer/heaplog_server
go build
./heaplog_server --logtostderr --document_root .. \
--binary=YOUR_APP_PATH/build/objs/fw.elf --console_log /tmp/console.log
```

Point your browser at <http://localhost:8910/> and press "connect".

This should give you a pretty picture like this:

API Reference

Repo	Description	Author
adc	ADC support	mongoose-os
arduino-adafruit-ads1x15	Arduino Adafruit ADS1015 library	
djzxz		
arduino-adafruit-bme280	Arduino Adafruit BME280 and BMP280 library	mongoose-os
arduino-adafruit-gfx	Arduino Adafruit GFX library	mongoose-os
arduino-adafruit-htu21df	Arduino Adafruit HTU21D-F library converted for Mongoose OS	pedalPusher68
arduino-adafruit-pwm-servo	Arduino Adafruit PWM servo library	
djzxz		
arduino-adafruit(ssd)1306	Arduino Adafruit SSD1306 library	mongoose-os
arduino-adafruit-tsl2561	Arduino Adafruit TSL2561 library converted for Mongoose OS	pedalPusher68
arduino-compat	Arduino compatibility	mongoose-os
arduino-dallas-temperature	Arduino DallasTemperature library	mongoose-os
arduino-ds3231	Arduino DS3231 library	
djzxz		
arduino-md-parola	Parola text effects library for MAX 7219	mongoose-os
arduino-onewire	Arduino OneWire library	mongoose-os
arduino-pololu-vl53l0x	Arduino pololu VL53L0X library	dcato daisuke.cato@gmail.com
arduino-spi	Arduino SPI library	mongoose-os
arduino-wire	Arduino Wire library	mongoose-os
atca	ATCA support	mongoose-os
aws	Amazon AWS IoT integration	mongoose-os
azure	Azure IoT Hub integration	mongoose-os
barometer	Library that implements various barometers	Pim van Pelt pim@google.com
bh1730	BH1730 sensor support	Angus Gratton
blynk	Blynk mobile app integration	mongoose-os
bm222	A Bosch BM222 accelerometer library	mongoose-os
bme280	Moongose OS BME280/BMP280 library	Liviu Nicolescu
bt-common	Bluetooth support	mongoose-os
bt-service-config	Configuration over Bluetooth GATT service support	mongoose-os
bt-service-debug	Configuration over Bluetooth GATT service support	mongoose-os
bt-service-time	Bluetooth Current Time Service implementation	mongoose-os

ca-bundle	CA Bundle	mongoose-os
ccs811-i2c	Mongoose OS library implementing an I2C driver for CCS811	Pim van Pelt pim@google.com
core	Core meta-lib	mongoose-os
cron	A Cron lib	mongoose-os
crontab	Crontab	mongoose-os
dallas-rmt	Implements Mongoose dallas rmt driver	nliviu
dash	Support library for the device management dashboard	mongoose-os
demo-bundle	Collection of libraries for demos	mongoose-os
dht	A DHT lib	mongoose-os
dns-sd	Implements DNS-SD protocol	mongoose-os
ds3231	DS3231 I2C library	Liviu Nicolescu nliviu@gmail.com
empty	An empty lib that does nothing	mongoose-os
esp32-touchpad	ESP32 touch sensor support	mongoose-os
ethernet	Ethernet support	mongoose-os
file-logger	File logger library	mongoose-os
gcp	Google IoT Core integration	mongoose-os
http-server	HTTP server for Mongoose OS	mongoose-os
htu21df-i2c	Mongoose OS library implementing an I2C driver for HTU21D(F)	Pim van Pelt pim@google.com
i2c	I2C support for Mongoose OS	mongoose-os
ili9341-spi	Mongoose-OS library for ILI9341	Pim van Pelt pim@google.com
imu	Library that implements various IMU devices	Pim van Pelt pim@ipng.nl
ir	IR protocol library	Vladimir Dronnikov dronnikov@gmail.com
js-demo-bundle	Collection of libraries for JavaScript demos	mongoose-os
jstore	JSON Store	mongoose-os
location	Location library	mongoose-os
mcp9808-i2c	Mongoose OS library implementing an I2C driver for MCP9808	Pim van Pelt pim@google.com
md-max72xx	Port of md-max72xx as a mongoose-os lib	mongoose-os
mjs	JavaScript engine	mongoose-os
mongoose	Mongoose Embedded Web Server Library	mongoose-os
mq135	MQ135 gas sensor support	mf.aoudia@gmail.com
mqtt	MQTT protocol support	mongoose-os

neopixel	Neopixel driver	mongoose-os
onewire	Implements 1-Wire API on Mongoose OS	mongoose-os
onewire-rmt	Implements Mongoose rmt onewire driver	nliviu
ota-http-client	Implements Mongoose OS OTA HTTP client	mongoose-os
ota-http-server	Implements Mongoose OS OTA HTTP server	mongoose-os
ota-shadow	OTA firmware update via the AWS IoT device shadow	mongoose-os
pppos	PPPoS support	mongoose-os
prometheus-metrics	Implements Mongoose Prometheus Metrics endpoint	Pim van Pelt pim@google.com
pwm	PWM support	mongoose-os
rpc-azure	RPC support for Azure cloud	mongoose-os
rpc-common	Implements Mongoose OS RPC over MQTT protocol	mongoose-os
rpc-gatts	RPC over Bluetooth GATT (server)	mongoose-os
rpc-loopback	Allows to call local RPC services - adds RPC Loopback channel	mongoose-os
rpc-mqtt	Implements Mongoose OS RPC over MQTT protocol	mongoose-os
rpc-service-atca	RPC ATCA Service support	mongoose-os
rpc-service-config	RPC Config Service support	mongoose-os
rpc-service-cron	RPC Service for Crontab	mongoose-os
rpc-service-fs	RPC Filesystem Service support	mongoose-os
rpc-service-gattc	RPC GATT Client Service	mongoose-os
rpc-service-gpio	RPC GPIO Service	mongoose-os
rpc-service-i2c	RPC I2C Service support	mongoose-os
rpc-service-ota	Support for Over-The-Air update via RPC	mongoose-os
rpc-service-wifi	RPC Wifi Service support	mongoose-os
rpc-uart	Implements Mongoose OS RPC over UART	mongoose-os
sensor-utils	A Mongoose OS lib of several formulas useful for developing sensor-based apps (temperature, pressure, etc.)	Bradley Smith bradley.1.smith@gmail.com , pedalPusher68
shadow	A cross-cloud device shadow API	mongoose-os
sht31-i2c	Mongoose OS library implementing an I2C driver for SHT31	Pim van Pelt pim@google.com
si7005	Si7005 Temperature and Humidity Sensor Driver	mongoose-os
si7021-i2c	Mongoose OS library implementing an I2C driver for Si7021	Pim van Pelt pim@google.com
sntp	A SNTP library	mongoose-os
spi	Implements SPI API on Mongoose OS	mongoose-os

ssd1306	Implements SSD1306 OLED driver on Mongoose OS	brandond
stmpe610-spi	STMPE610 SPI based touch screen	Pim van Pelt pim@google.com
tmp006	A Texas Instruments TMP006 temperature sensor	mongoose-os
veml6075-i2c	Mongoose OS library for VEML6075	Pim van Pelt pim@google.com
vfs-common	VFS common bits	mongoose-os
vfs-dev-ram	Implements RAM-based VFS device	mongoose-os
vfs-dev-spi-flash	VFS device driver for SPI NOR flash	mongoose-os
vfs-fs-spiffs	Implements the mos VFS interface for SPIFFS	mongoose-os
watson	IBM Watson IoT Platform support	mongoose-os
wifi	WiFi support	mongoose-os

Configuration infrastructure

Mongoose OS uses a structured, multi-layer configuration.

It consists of two parts: a compile time part that defines configuration, and a run time part that uses configuration.

Compile-time generation

- Any piece of code that requires config setting, can define a .yaml file that describes configuration parameters for that piece of code
- User code could define it's own set of configuration parameters in it's own .yaml file
- All these yaml files are merged together during firmware compilation, and a single `sys_config_defaults.json` file is generated
- User-defined YAML file is applied last, therefore it can override any default settings specified in the system .yaml files
- Generated `sys_config_defaults.json` file represents all possible configurable settings for the firmware
- A C header and source files are also generated. C header contains a structure that mirrors `sys_config_defaults.json` file, and an API for getting and setting individual configuration values

Run-time init

- `conf0.json` - configuration defaults. This is a copy of the generated `sys_config_defaults.json`. It is loaded first and must exist on the file system. All other layers are optional.
- `conf1.json` - `conf8.json` - these layers are loaded one after another, each successive layer can override the previous one (provided `conf_acl` of the previous layer allows it). These layers can be used for vendor configuration overrides.
- `conf9.json` is the user configuration file. Applied last, on top of all other layers. `mos config-set` and `save_cfg()` API function modify `conf9.json`.

Therefore here are the rules of thumb:

- If you need to define your own config parameters, add a `config_schema`

section in the `mos.yml` file, as described in the [quick start guide](#)

- If you want to override some system default setting, for example a default UART speed, also use `config_schema` and add overrides there, see [example](#)
- If you want to put some unique information on each firmware, for example a unique ID, and optionally protect it from further modification, use any of the layers 1 through 8, e.g. `conf5.json`.
- `conf9.json` should never be included in the firmware, or it will override user's settings during OTA.

So, firmware configuration is defined by a set of YAML description files, which get translated into an opaque C structure `mgos_sys_config` and public accessors during firmware build: getters like `mgos_sys_config_get_....()` and setters like `mgos_sys_config_set_....(value)`. C code can access configuration parameters by invoking those accessors. Fields can be integer, boolean or string. C functions to retrieve and save that global configuration object are generated.

Example on how to access configuration parameters:

```
printf("My device ID is: %d\n", mgos_sys_config_get_device_id()); // Get config param
```

Example on how to set configuration parameter and save the configuration:

```
mgos_sys_config_set_debug_level(2); // Set numeric value
mgos_sys_config_set_device_password("big secret"); // Set string value
char *err = NULL;
save_cfg(&mgos_sys_config, &err); /* Writes conf9.json */
printf("Saving configuration: %s\n", err ? err : "no error");
free(err);
```

The generation mechanism not only gives a handy C API, but also guarantees that if the C code accesses some parameter, it is indeed in the description file and thus is meant to be in the firmware. That protects from the common problems when the configuration is refactored/changed, but C code left intact.

Mongoose OS configuration is extensible, i.e. it is possible to add your own configuration parameters, which might be either simple, or complex (nested).

At run time, a configuration is backed by several files on a filesystem. It has multiple layers: defaults (0), vendor overrides (1-8), and user settings (9). Vendor layers can "lock" certain parts of configuration for the user layer, and allow only certain fields to be changed. For example, end-user might change the WiFi settings, but cannot change the address of the cloud backend.

Compile time generation deep dive

Configuration is defined by several YAML files in the Mongoose OS source repository. Each Mongoose OS module, for example, crypto chip support module, can define its own section in the configuration. Here are few examples:

- `mgos_sys_config.yaml` is core module, defines debug settings, etc
- `mgos_atca_config.yaml` is a crypto chip support module
- `mgos_mqtt_config.yaml` has default MQTT server settings

As has been mentioned in the overview, you can define your own sections in the config, or override existing default values. This is done by placing a

config schema descriptor into `mos.yml`, like this:

```
config_schema:  
  - ["hello", "o", {"title": "Hello app settings"}]  
  - ["hello.who", "s", "world", {"title": "Who to say hello to"}]
```

When the firmware is built, all these YAML files get merged into one.

User-specified YAML file goes last, therefore it can override any other.

Then, merged YAML file gets translated into two C files, `mgos_config.h` and `mgos_config.c`. You can find these generated files in the `YOUR_FIRMWARE_DIR/build/gen/` directory after you build your firmware.

Here's a translation example of a custom `src/conf_schema.yaml`:

```
[  
  ["hello", "o", {"title": "Hello app settings"}],  
  ["hello.who", "s", "world", {"title": "Who to say hello to"}]  
]
```

It gets translated into the following getter and setter:

```
const char *mgos_sys_config_get_hello_who(void);  
void mgos_sys_config_set_hello_who(const char *val);
```

Then, C firmware code can access that custom configuration value:

```
printf("Hello, %s!\n", mgos_sys_config_get_hello_who());
```

Numbers are represented by integers in C, as are booleans.

Strings will be allocated on the heap.

IMPORTANT NOTE: Empty strings will be represented as `NULL` pointers, be careful.

Currently, all substructs are actually public and can be retrieved with their own getter; thus the header contains the struct definition and the getter:

```
struct mgos_config_hello {  
    char *who;  
};  
  
const struct mgos_config_hello *mgos_sys_config_get_hello(void);
```

It's useful to have universal functions which take the whole struct as a parameter. In the future though there will be an option to make some particular struct public, and by default all structs will be private.

Run time - factory, vendor, user layers

Device configuration is stored on the filesystem in several files:

- `conf0.json` - factory defaults layer
- `conf1.json` to `conf8.json` - vendor layers
- `conf9.json` - user layer

When Mongoose OS boots, it reads those files in exactly that order, merges into one, and initializes in-memory C configuration structure reflects that on-flash configuration. So, at boot time, `struct mgos_config` is initialised in the following order:

- First, the struct is zeroed.
- Second, defaults from `conf0.json` are applied.
- Third, *vendor configuration layers* 1 through 8 are loaded one after another.
- The *user configuration file*, `conf9.json`, is applied on as the last step.

The result is the state of the global struct `mgos_config`.

Each step (layer) can override some, all or none of the values.

Defaults must be loaded and it is an error if the file does not exist at the time of boot. But, vendor and user layers are optional.

Note that a vendor configuration layer is not present by default.

It is to facilitate post-production configuration: devices can be customised by uploading a single file (e.g. via HTTP POST to `/upload`) instead of performing a full reflash.

Vendor configuration is not reset by the "factory reset", whether via GPIO or web.

Field access control

Some settings in the configuration may be sensitive and the vendor may, while providing a way for user to change settings, restrict certain fields or (better) specify which fields can be changed by the user.

To facilitate that, the configuration system contains field access control, configured by the **field access control list** (ACL).

- ACL is a comma-delimited list of entries which are applied to full field names when loading config files at boot time.
- ACL entries are matched in order and, search terminates when a match is found.
- ACL entry is a pattern, where `*` serves as a wildcard.
- ACL entry can start with `+` or `-`, specifying whether to allow or deny change to the field if the entry matches. `+` is implied but can be used for clarity.
- The default value of the ACL is `*`, meaning changing any field is allowed.

ACL is contained in the configuration itself - it's the top-level `conf_acl` field. The slight twist is that during loading, the setting of the *previous* layer is in effect: when loading user settings, `conf_acl` from vendor settings is consulted, and for vendor settings the `conf_acl` value from the defaults is used.

For example, to restrict users to only being able change WiFi and debug level settings, `"conf_acl": "wifi.*,debug.level"` should be set in `conf{1-8}.json`.

Negative entries allow for default-allow behaviour:

`"conf_acl": "-debug.*,*"` allows changing all fields except anything under `debug`.

Resetting to factory defaults

If configured by `debug.factory_reset_gpio`, holding the specified pin low during boot will wipe out user settings (`conf9.json`).

Note, vendor settings, if present, are not reset.

Introduction

This overview is about the design of

[Mongoose OS](#), a firmware development framework for connected products. If you are an IoT firmware developer, Mongoose OS is for you.

Here we share our vision and the rationale for the design decisions we made. The vast majority of these decisions were driven by our work for our customers, when we developed device firmware to bring their IoT products to the market.

We noticed the following:

1. Each project was a start from scratch, more or less
2. Only ~10-20% of the time was spent on the actual product-specific logic.
The rest was a generic infrastructure - like configuration, OTA, etc

We refactored those generic pieces that take up to 90% of firmware development time, into a reusable set of components. We made it platform-independent - for example, toggling a GPIO code on Mongoose OS looks the same on all hardware platforms.

The result we called **Mongoose OS**.

Where does the name Mongoose come from?

We are targeting IoT products, where networking is crucial. We use a mature and trusted

[Mongoose Networking Library](#) as

the networking core - that is the origin of the name. The networking library uses `mg_` prefix for all API functions, and similarly Mongoose OS uses `mgos_` prefix.

Our goal is to share our experience in the hope that it'll help other developers to save a great deal of time and effort, reusing a solid and reliable basis for their products.

Architecture

Components

Mongoose OS is a framework for building apps (firmwares) for low-power microcontrollers (uC), and consists of the following main components:

- A `mos` tool. Provides device management and firmware building capabilities
- A build toolchain. This is a docker image which contains hardware vendor's SDK together with mongoose-os sources. A `mos build` command builds a firmware (we call it an "app") by taking `mos.yml` file in the current directory and invoking a build docker image either remotely (`mos build`) or locally (`mos build --local`).
- A collection of ready-to-use **apps** and **libraries**

Mongoose OS is based on the vendor's SDK, it extends the capabilities of the native SDK. For example, on ESP32 uC, Mongoose OS uses an ESP-IDF SDK, therefore it provides all capabilities that ESP-IDF provides, plus extra that come with Mongoose OS. If user code uses crossplatform API only, it can be built on all supported hardware platforms with no code changes:

If we zoom in the yellow "Mongoose OS" block, it is fragmented into several components as well. Some of them, like configuration, RPC, timers,

networking API, etc, will be covered further down.

Source code structure

The Mongoose OS core lives at [cesanta/mongoose-os](https://github.com/cesanta/mongoose-os) on GitHub:

- `src/` - cross-platform API
- `src/common/` - various utility functions
- `src/frozen/` - JSON parser/emitter
- `platforms/` - platform-specific code

The bulk of the functionality, however, is split into libraries. Each library is a separate GitHub repository, collected under the [mongoose-os-libs](https://github.com/mongoose-os-libs) organisation, which serves as a central repository of libraries.

When documentation is generated, all libraries are traversed and the "API Reference" part is automatically generated. The `docs:` tag in the library's `mos.yml` file specifies the documentation category and title. For example, for the ADC library located at <https://github.com/mongoose-os-libs/adc>, the `mos.yml` contains:

```
tags:  
  - docs:core:ADC  
  ...
```

That creates an [API Reference/Core/ADC](#) documentation page. The content is generate from the [README.md](#) and header files.

Boot process

The boot process is driven by a cross-platform

`mgos_init.c`.

In short, the subsystems are initialised in the following order:

Native SDK init, GPIO, configuration, WiFi, platform-specific init, libraries (they can define their initialisation order), user app init function `mgos_app_init()`, and at the end - all registered `MGOS_HOOK_INIT_DONE` hooks are invoked.

The initialisation function has the following prototype:

```
enum mgos_init_result mgos_XXX_init(void);
```

It returns `MGOS_INIT_OK` on success, or any other specific numeric code on error.

If any of those init functions returns an error, the firmware reboots immediately. This is done intentionally, in order to revert back to the previous firmware in case of failed OTA update.

File system

Mongoose OS implements Virtual File System layer, VFS. That means it can attach (mount) different storage types into a single file system tree.

For example, a device can have an SPI flash storage and an SD card storage. For each storage type, a filesystem driver must be implemented. For example, it is possible to write a driver that implements a Dropbox or Google Drive storage type, and a device (e.g. ESP8266 module) can mount a Dropbox folder.

Mongoose OS provides a

[Filesystem RPC service](#)

that allows remote filesystem management - for example, you can edit files remotely.

The contents of the filesystem depends on the app and specific libraries that are used. For example, an `mjs` library which adds JavaScript support to the app, also adds a bunch of `api_*.js` files to the filesystem. Here is a typical layout:

```
conf0.json      - default app configuration, must NOT be edited manually
conf9.json      - user-specific overrides, changed by "mos config-set" command
index.html     - many apps define this file, which is served by a web server
ca.pem         - added by the ca-bundle library, contains ca root certs
init.js        - main file for the JavaScript-enabled apps
api_*.js       - JavaScript API files
api_*.jsc      - compiled JavaScript files
```

Main event loop

Mongoose OS contains

[Mongoose Networking Library](#) as one

of the core components. The networking library provides network protocol support, such as UDP, MQTT, etc. It constitutes the low level of Mongoose OS; it is non-blocking and event based, uses `mg_` API prefix and expects the following usage pattern:

- User creates listening connections by calling `mg_bind()` or variants
- User creates outgoing connections by calling `mg_connect()` or variants
- For all connections, listening or outgoing, user must define an event handler function
- All connections are inserted in a linked list in a `struct mg_mgr` structure, which is an event manager
- User makes an infinite event loop. On each iteration, Mongoose Networking library waits for IO on all connections. When it happens, an event handler is called for a appropriate connection

Mongoose OS does exactly that. It defines a "system" event manager instance, and runs a main event loop in a single task. That event loop dispatches events by calling event handlers. For example, `mgos_gpio_set_button_handler()` function sets up a button press event handler. When a hardware interrupt occurs, its handler queues the event, and the Mongoose OS task calls the user-defined button handler in its context.

For network connections, Mongoose OS defines wrappers for low-level `mg_` functions. These wrappers use "system" event manager and provide the reconnection functionality for the outgoing connection.

For example, low-level `mg_` API for MQTT protocol allows to create an MQTT client. If it disconnects for any reason, e.g. temporary WiFi connectivity loss, the connection closes. The `mgos_` wrapper, however, would setup a reconnection

timer with exponential backoff and re-establish the connection automatically.

This is a valuable addon to the low-level mg_ API, therefore using

mgos_ API is a good idea. Of course the low level mg_ API is also available.

You can get main event manager instance by calling `mgos_get_mgr()`

function defined in `mgos_mongoose.h`.

Note that mgos_ API, as well as mg_ API, is cross-platform. A firmware written with that API only, is portable between supported architectures, as demonstrated by many [example apps](#).

However, the native SDK API is not in any way hidden and is fully available.

For example, one could fire extra FreeRTOS tasks on platforms whose SDK use FreeRTOS. The price to pay is loss of portability.

Example footprint

Mongoose OS is highly modular - it is possible to include or exclude functionality depending on specific needs. That is implemented by the library mechanism, described later. In order to get a feeling about the resulting footprint, the table below shows measurements done on

[TI CC3220SF](#)

platform for Mongoose OS 1.18 release, built with different options.

RAM figures are measured after Mongoose OS is initialised, i.e. those numbers are what is available for the application code.

Name	Code Size	Free RAM	Notes
minimal	113k	230k	An example-no-libs-c app. Includes RTOS, TCP/UDP networking core, file system, configuration infrastructure, SNTP
minimal+mjs	160k	229k	Minimal + mJS JavaScript engine
minimal+aws	133k	230k	Minimal + AWS IoT support
minimal+gcp	159k	230k	Minimal + Google IoT Core support
js-demo	304k	225k	A default JS demo app . Includes HTTP, MQTT, WebSocket, mDNS networking, RPC subsystem, AWS IoT, Google IoT Core, JavaScript, I2C, SPI, PWM, ADC, and more - see js-demo-bundle library

RPC - Remote Procedure Calls

RPC means Remote Procedure Call. This is the way to send commands to devices and receive replies, i.e. call remote procedures.

Mongoose OS uses

[JSON-RPC 2.0](#).

Since the RPC mechanism uses JSON-RPC, that means that the "procedure", or an "RPC service" that device implements, is a function written in C/C++ or JavaScript with the following properties:

- Has a name, for example `GPIO.Toggle`,
- Takes a `params` JSON object,
- Returns a `result` JSON object.

For example, an RPC request to set GPIO pin 2 to high voltage looks like this:

```
{ "id": 1932, "method": "GPIO.Write", "params": {"pin": 2, "value": 1} }
```

The successful reply from a device looks like this:

```
{ "id": 1932, "result": true }
```

The failure reply looks like this:

```
{ "id": 1932, "error": { "code": 400, "message": "error setting pin mode" } }
```

Note that the "jsonrpc": "2.0" attribute in the request frame can be omitted.

Mongoose OS libraries implement a large set of ready-to-go RPC services, like managing hardware peripherals (GPIO, SPI, I2C), managing files, remote updates, etc. It is easy to add custom RPC services too, see [RPC Core library](#) for more details.

The JSON-RPC messages could be carried out by many different channels: serial (UART), HTTP/Restful, WebSocket, MQTT, Bluetooth. RPC API allows to add support for other channels. `mos` tool provides an easy way to call device's RPC services over the serial connection or over the network. Alternatively, RPC services can be invoked programmatically.

That means, you can write a function in C or JavaScript, and call it remotely via multiple protocols. Example: an RPC service `Sum` that adds two numbers `a` and `b`:

```
RPC.addHandler('Sum', function(args) {
  if (typeof(args) === 'object' && typeof(args.a) === 'number' && typeof(args.b) === 'number') {
    return args.a + args.b;
  } else {
    return {error: -1, message: 'Bad request. Expected: {"a":N1,"b":N2}'};
  }
});
```

See [RPC.addHandler API reference](#).

The C/C++ implementation that does the same would look something like this:

```
#include "mgos_rpc.h"

static void sum_cb(struct mg_rpc_request_info *ri, void *cb_arg,
                   struct mg_rpc_frame_info *fi, struct mg_str args) {
  double a = 0, b = 0;
  if (json_scanf(args.p, args.len, ri->args_fmt, &a, &b) == 2) {
    mg_rpc_send_responsef(ri, "%lf", a + b);
  } else {
    mg_rpc_send_errorf(ri, -1, "Bad request. Expected: {"a":N1,"b":N2}");
  }
  (void) cb_arg;
  (void) fi;
}

// Somewhere in init function, register the handler:
mg_rpc_add_handler(mgos_rpc_get_global(), "Sum", "{a: %lf, b: %lf}", sum_cb, NULL);
```

See [mgos_rpc_add_handler API reference](#). Make sure to include the `rpc-common` library in your app's `mos.yml`.

Here is how you can call it:

Via serial connection, if a device is attached to a serial port:

```
mos --port /dev/ttyUSB0 call Sum '{"a":1, "b": 2}'  
3
```

Via WebSocket, if a device is in the local network:

```
mos --port ws://192.168.0.206/rpc call Sum '{"a":1, "b": 2}'  
3
```

Via RESTful call, if a device is in the local network:

```
curl -d '{"a":1, "b": 2}' 192.168.0.206/rpc/Sum  
3
```

Via an MQTT server, talking to an device idendified by its ID esp32_6732ac:

```
mos --port mqtt://my.mqtt.server:1883/esp32_6732ac call Sum '{"a":1, "b": 2}'  
3
```

To see a list of all RPC services implemented by a device, call `RPC.List`:

```
mos call RPC.List  
[  
    "I2C.WriteRegW",  
    "I2C.WriteRegB",  
    "I2C.ReadRegW",  
    "I2C.ReadRegB",  
    ...  
]
```

Remote management

Mongoose OS has many built-in RPC services: for managing files, accessing hardware peripherals, performing over-the-air updates, etc. In fact, the majority of `mos` command talks to a device via an RPC call.

See [Secure remote device management with Mongoose OS and AWS IoT](#) for an example of remote management over the secure AWS IoT cloud. In that case, Mongoose OS uses secure MQTT RPC channel, authenticated via the mutual TLS - per AWS IoT standards.

An ability to call RPC services via network add an incredible level of control, implementing remote management capabilities at the highest levels.

Example: scan I2C bus and return addresses of I2C peripherals from a device at IP address 192.168.1.4 using `mos` tool in a command line mode:

```
mos --port ws://192.168.1.4/rpc call I2C.Scan  
[  
    31  
]
```

Licensing

Mongoose OS is highly modular. It consists of the [core](#)

and a large number of [libraries](#). The core, and most of the libraries, are open source and released under the [Apache 2.0 license](#) as Mongoose OS Community Edition.

Some of the libraries are closed-source, and have certain restrictions. Those restrictions can be removed by purchasing a Mongoose OS Enterprise Edition license.

Mongoose OS Enterprise Edition Licensing process

- Make sure that `rpc-service-ota` library is used by your app
- Login to the [Mongoose License Manager](#)
- Buy any number of licenses you need. IMPORTANT: choose "mongoose-os" license type
- If your device is connected to your workstation over USB, run:

```
mos license
```

- Or, if your device is accessible remotely, run:

```
mos --port DEVICE_ADDRESS license
```

Licenses are bound to devices, and they are permanent.

The `mos license` command creates a unique license string specific for a device, and updates `device.license` configuration parameter, which "unlocks" the device and removes restrictions. The License Manager decreases the number of available licenses and saves a unique license string. The next time `mos license` is run for the same device, an old license string is returned by the License Manager but the number of available licenses is not decreased. Thus a device, once licensed, is licensed permanently.

Libraries with restrictions

Below is the list of closed-source libraries and their restrictions.

Library	Restriction
ota-common	OTA only from mDash
ota-http-client	OTA only from mDash
ota-http-server	No restrictions
ota-shadow	OTA only from mDash
cron	3 cron tasks max
crontab	3 crontab entries max

Apps and libraries

A Mongoose OS **app** is a firmware that does something specific. It could be built and flashed on a microcontroller. For example, a [blynk app](#) is a firmware that makes a device controllable by the [Blynk mobile app](#).

Another example is a [default app](#)

that gets flashed when you press a "Flash" button on a Web UI "device control" dialog, or type `mos flash <arch>` on a terminal. That default app blinks an LED periodically, can talk to an MQTT server, and allows user to extend the logic by editing JavaScript code directly on a device filesystem.

An app can use any number of libs. A **lib** is a reusable library. It cannot be built directly into a working firmware, because it only provides an API but does not actually use that API. An app can include a lib by listing it in the `libs:` section of the [mos.yml](#) file.

`mos build` command generates code that calls library initialisation functions. Libraries are initialised in the order of their reference.

Local and remote builds

By default, a `mos build` command that builds an app's firmware, is using so-called remote build - it packs app's sources and sends them over to the Mongoose OS build machine. This is the default behavior, cause it does not require a [Docker](#) installation on the workstation.

However, if a Docker is installed, then it is possible to build locally. This is done by adding an extra `--local` flag (see below). In this case, everything is done on the local machine. This is a preferable option for the automated builds, and for those who do not want their sources leaving their workstations. Summary:

Build type	Build command
Remote (default)	<code>mos build --platform PLATFORM</code>
Local (requires Docker)	<code>mos build --platform PLATFORM --local --verbose</code>

mos.yml file format reference

`mos.yml` file drives the way Mongoose apps are built. Below is a description of the sections (keys) in this file. Libraries also have `mos.yml` files, the only difference with apps is that they have `type: lib` key and they cannot be built into a firmware. So the following applies to both apps and libraries.

author

A string, `FirstName SecondName <Email>` of the author, example:

```
author: Joe Bloggs <joe@bloggs.net>
```

build_vars

List of Makefile variables that are passed to the architecture-specific Makefile when an app is getting built. See next section for a build process deep-dive. An example of arch-specific Makefile is: [platforms/esp32/Makefile.build](#).

The others are in the respective directories: `fw/platforms/*/Makefile.build`.

The example below changes ESP32 SDK configuration by disabling brownout detection:

```
build_vars:  
  ESP_IDF_SDKCONFIG_OPTS: "${build_vars.ESP_IDF_SDKCONFIG_OPTS} CONFIG_BROWNOUT_DET="
```

Another example is the [dns-sd library](#) that enables DNS-SD:

```
build_vars:  
  MGOS_ENABLE_MDNS: 1
```

binary_libs

A list of `.a` libs or directories with those. Do not put trailing slashes to directory names:

```
binary_libs:  
  - mylib/mylib.a
```

cdefs

Additional preprocessor flags to pass to the compiler, example:

```
cdefs:  
  FOO: BAR
```

That gets converted into the `-DFOO=BAR` compilation option, for both C and C++ sources.

cflags, cxxflags

Modify compilation flags for C (`cflags`) and C++ (`cxxflags`). For example, by default warnings are treated as errors. This setting ignores warnings when compiling C code:

```
cflags:  
  - "-Wno-error"
```

If what you're after is defining preprocessor variables, `cdefs` makes it easier. This snippet:

```
cdefs:  
  FOO: BAR
```

Is the same as:

```
cflags:  
  - "-DFOO=BAR"  
cxxflags:  
  - "-DFOO=BAR"
```

config_schema

This can define a new configuration section for the device, and also override a previously defined configuration entries defined elsewhere. For example, the following snippet defines a new section `foo` and overrides a default

value of `mqtt.server` set by the `mqtt` library:

```
config_schema:  
  - ["foo", "o", {title: "my app settings"}]  
  - ["foo.enable", "b", true, {title: "Enable foo"}]  
  - ["mqtt.server", "1.2.3.4:1883"]
```

description

A string, one-line short description, example:

```
description: Send BME280 temperature sensor readings via MQTT
```

filesystem

A list of files or directories with files to be copied to the device's

filesystem, example:

```
filesystem:  
  - fs  
  - other_dir_with_files  
  - foo/somepage.html
```

includes

A list of directories with C/C++ include files. Do not put trailing slash

to the directory name. Example:

```
includes:  
  - my_stuff/include
```

libs

Library dependencies. Each library should have an `origin` and optionally can

have `name` and `version`. `origin` is a GitHub URL, like

<https://github.com/mongoose-os-libs/aws> (note: it must be a repo with `mos.yml` in the repo root!).

`Name` is used to generate the code which calls

library initialization function: e.g. if the lib name is `mylib`, it should have the function `bool mgos_mylib_init(void)`. Also, for local builds, `name` is used as a directory name under `deps`: that's where `mos` clones libraries.

`version` is a git tag name, or branch name, or SHA of the library's repository. If omitted, it defaults to the `libs_version` in `mos.yml`, which, in turn, defaults to the `mos` tool version. So e.g. if the `mos` tool version is `1.21`, then by default it will try to use libs with the tag `1.21`. Latest `mos` will use the `master` branch.

Example:

```
libs:  
  # Use aws lib on the default version  
  - origin: https://github.com/mongoose-os-libs/aws  
  
  # Use aws lib on the version 1.20  
  - origin: https://github.com/mongoose-os-libs/aws  
    version: 1.20  
  
  # Use the lib "mylib" located at https://github.com/bob/mylib-test1  
  - origin: https://github.com/bob/mylib-test1  
    name: mylib
```

name

Override app or lib name. By default, the name is set equal to the directory name.

```
name: my_cool_app
```

sources

A list of C/C++ source files or directories with those. Do not put trailing slashes to directory names:

```
sources:  
  - src  
  - foo/bar.c
```

tags

A list of free-form string tags, used for Web UI search.

Some tags are predefined, they place the app or library in a certain category.

Those predefined tags are: `cloud` (cloud integrations),
`hardware` (hardware peripherals or API),
`remote_management` (remote management), `core` (core functionality). Example:

```
tags:  
  - cloud  
  - JavaScript  
  - AWS
```

Build process deep dive

When `mos build [FLAGS]` command is executed in the app directory, the following happens:

- `mos` scans `libs:` section of the `mos.yml` file and imports all libraries into the `libs` directory (`~/.mos/libs`, could be overridden by `--libs-dir ANOTHER_DIR` flag)
- Each library also has `mos.yml` file, and a library could have a `libs:` section as well - this way the library can depend on other library. `mos` imports all dependent libraries too, recursively.
- When all required libraries are imported, `mos` executes `git pull` in each of them, in order to update. That could be switched off by `--no-libs-update` flag.
- At this point, all required libraries are imported and updated.
- `mos` combines app's `mos.yml` file together with the `mos.yml` files of

all dependent libraries, merging them into one file. The order of merging

is this: if `my-app` depends on library `lib1`, and library `lib1` depends

on library `lib2`, then

`result_yml = lib2/mos.yml + lib1/mos.yml + my-app/mos.yml`. Meaning, the application's `mos.yml` has the highest priority.

- If `--local --verbose --repo PATH/TO/MONGOOSE_OS_REPO` flag is specified, then `mos` starts a local build by invoking `docker.cesanta.com/ARCH-build` docker image. That image encapsulates a native SDK for the given architecture together with Mongoose OS sources, <https://github.com/cesanta/mongoose-os>.
`mos` tool invokes `make -f fw/platforms/ARCH/Makefile.build` for the given platform. The result of this docker invocation is a `build/` directory with build artifacts and `build/fw.zip` firmware zip file which could be flashed to the device with `mos flash` command.
- If `--local` flag is not specified, packs source and filesystem files and sends them to the Mongoose OS cloud build backend at <http://mongoose.cloud>, which performs an actual build as described in the previous step, and sends back a `build/` directory with built `build/fw.zip` and artifacts.
- Generated artifacts in the `build/` directory is as follows:

```
build/fw.zip - a built firmware  
build/fs      - a filesystem directory that is put in the firmware  
build/gen     - a generated header and source files
```

How to create a new library

- The best way to develop a new library is as part of an app development. In your app, do a local build, which creates a `deps/` directory. That is the directory where you should place your new library.
- Clone an `empty` library, which is a skeleton for the new library, into the `deps/mylib` directory (change `mylib` to whatever name you want):
`git clone https://github.com/mongoose-os-libs/empty deps/mylib`
- Create `include/mgos_mylib.h` and `src/mgos_mylib.c` files in your library:

mgos_mylib.c:

```
#include "mgos_mylib.h"

// NOTE: library init function must be called mgos_LIBNAME_init()
bool mgos_mylib_init(void) {
    return true;
}
```

mgos_mylib.h:

```
#include "mgos.h"
```

- You can add your library-specific API to `mgos_mylib.h` and implementation in `mgos_mylib.c`.
- In your app's `mos.yml` file, add a reference to the new library:

```
libs:  
  - name: mylib
```

- Click on build button to build an app, and flash button to flash it
- Edit library source files `mylib/src`, build `myapp` until a test app

works as intended.

How to port an Arduino library

- Follow the steps outlined in the previous section.
- Copy Arduino library sources into the `mylib/src` directory, and `.h` files into the `include/` directory
- Add C wrappers for the C++ API. Make it so that wrappers are FFI-able into JS : use simple types in the API, max 6 32-bit params, 2 64-bit params. See <https://github.com/cesanta/mjs#cc-interoperability>
- If you plan to add JavaScript support too, create `mjs/api_mylib.js` file with the FFI JS wrappers.
- Build / test `myapp` until it works.
- See example libraries at <https://mongoose-os.com/docs/mongoose-os/api/arduino/arduino-adafruit-ads1x15.md>

Contributing an app or library

If you would like to share your project with a community and publish

it under the [Apache 2.0 license](#),

please follow these steps:

- Build your app as described in the previous section, flash and test it.
- Modify `mos.yml`, set `author` field as `Your Name <your@email.address>`.
- Make sure you have a descriptive `README.md` file.
- If this is a library:
 - Create `mjs_fs/api_<name>.js` file if your library has JavaScript API.
 - If it is a port of an Arduino library, make sure you include `arduino-compat` library in `mos.yml` file, see [arduino-adafruit-ssd1306 lib](#) for an example
 - See <https://github.com/mongoose-os-libs/blynk> for the reference
 - Consider contributing an example app that uses your library
- [Start a new discussion on forum](#) with a subject `New contribution: ...`, show a link to your code on GitHub / Bitbucket / whatever, or attach a zip file with the app sources.

Commercial firmware

Below is the procedure on how to build a firmware using

Mongoose OS Enterprise Edition (commercial version).

1. Acquire Mongoose OS license
2. Nominate a single GitHub account (user, not organisation) that gets an access to the private repo <https://gitub.com/cesanta/mos-libs>
3. On your build machine,
 - Clone `mos-libs` to some directory
 - Add `--local --verbose --libs-dir /path/to/mos-libs` arguments to your `mos build` command

Example command line to build a firmware:

```
mos build --platform stm32 --build-var BOARD=B-L475E-IOT01A \
--local --verbose --libs-dir ../mos-libs
```

Mongoose OS public libraries are located under the

<https://github.com/mongoose-os-libs> organisation. That org contains libraries authored by Cesanta, as well as libraries contributed by the community.

The mos-libs private repository contains only libraries authored by Cesanta. Also,

the mos-libs repository contain the source code for all libraries, whereas

some of the mongoose-os-libs libraries are in the binary form, built from sources with certain compilation flags that enable restrictions.

The commercial firmware must be built against the mos-libs libraries, in order to use the sources, without enabling restrictions. When you specify `--libs-dir /path/to/fetched/mos-libs` then your firmware is built correctly: Cesanta's libraries would use the locally fetched mos-libs, and all other libraries clone mongoose-os-libs.

- [Introduction](#)
- [RPC mechanism](#)
- [Device config](#)
- [Build Process](#)
- [Debugging](#)
- [OTA updates](#)
- [Security](#)
- [mos tool reference](#)
- [Licensing](#)

mbedTLS library optimisation

Mongoose OS uses mbedTLS library from ARM with several patches:

- RAM usage by each connection. By default, mbedTLS uses allocates >32k RAM for each TLS connection. Mongoose OS uses dynamic buffers, reducing RAM usage per connection down to 1k RAM.
- ATECC608A crypto chip integration. This patch makes mbedTLS to offload crypto to the hardware chip.
- Dynamic CA certificates loading. By default, mbedTLS loads all CA certificates in RAM. Our patch makes it load on demand, saving a lot of RAM. Mongoose OS stores CA certificates in ca.pem file, where you can add your server's CA certificate without blowing RAM usage.

ESP32 flash encryption

Encrypting on-flash code and data is essential:

- it protects from the reverse engineering
- it protects the device credentials, like private keys, being copied and used to get cloud access

ESP32 chip comes with built-in security features, one of which is a

transparent SPI flash encryption - for details, see

[Espressif documentation](#).

Mongoose OS makes ESP32 flash encryption setup easy.

By default, Mongoose OS firmware is flashed in plain-text form:

```
mos flash esp32          # Flash Mongoose OS on ESP32
mos flash-read --arch esp32 0x190000 2000 -  # Dump filesystem area
```

The `flash-read` command dumps the flash memory into a file, and the output can show that the contents is not encrypted. Therefore, sensitive information like TLS private keys could be easily stolen from the flash. In this case, we see a part of the device's file system, not encrypted.

In order to enable flash encryption, use `esp32-gen-key` command. It enables flash encryption for the next flashing (and sets efuses):

```
mos -X esp32-gen-key flash_encryption_key fe.key \
--esp32-enable-flash-encryption --dry-run=false
mos flash esp32 --esp32-encryption-key-file fe.key
```

That is irreversible - once flash encryption is enabled, you cannot go back.

Note the extra flag `--esp32-encryption-key-file fe.key`

for the `flash` command. From now on, a key file is required to re-flash the device.

If the key file is lost, the module can't be reflashed.

After flash encryption is enabled, the very first boot performs an encryption, which takes a while - up to a minute.

Subsequent boots will be normal, not doing any encryption.

Once the flash is encrypted, one can verify it using `flash-read` command to ensure there no plain-text parts are present:

```
mos flash-read --arch esp32 0x190000 2000 -
```

It is recommended to use a separate key for each device. The efuses must be set to enable encryption (this is done automatically by `esp32-gen-key`) and can be verified by running:

```
mos esp32-efuse-get
```

A device with encryption enabled should show `flash_crypt_cnt : 0x01`

ATECC608A crypto chip

Often, IoT boards provide no built-in flash protection mechanism.

Anyone with a physical access to the device can read the whole flash, including any sensitive information like TLS private keys.

Crypto chips are designed to mitigate that.

Their main function is provide storage for private keys, which cannot be read.

Private keys are stored inside the crypto chip, and all the crypto operations that require private key, are offloaded to the crypto chip which performs the operation and gives the result back.

ATECC608A crypto chip

is designed with additional hardware protection mechanisms to make key extraction difficult. It is an impressive piece of hardware with many layers of protection, and important enough it is quite inexpensive, costing less than 80 cent a piece.

Wiring (ESP8266 NodeMCU example)

Get ATECC608A - either as an ATCRYPTOAUTH-XPRO board which requires no soldering, or a bare-bones ATECC608A which requires soldering.

Function	ATECC608A pin	ESP8266 pin	NodeMCU pin	ATCRYPTOAUTH pin
SDA	5	10 (GPIO12)	D6	11 (yellow)
SCL	6	9 (GPIO14)	D5	12 (white)
GND	4	Any GND	Any GND	19 (black)
VCC	8	Any 3V3	Any 3V3	20 (red)

Wiring for ATCRYPTOAUTH-XPRO:

Wiring for the bare-bones ATECC608A:

Setup guide

Mongoose OS has native support for
[ATECC608A](#) security chip.

This section is a quick guide to get it up and running.

For a more detailed reference, especially of chip configuration, please refer to Microchip documentation.

1. The chips leave the factory unconfigured. Blank chip will be detected but crypto operations failing with code `0xf4`. Microchip provides their own configuration tools but `mos` includes basic commands to get and set configuration as a YAML file. For development you can use our [sample configuration](#). To set it, use the following `mos` commands:

```
mos atca-set-config atca-test-config.yaml --dry-run=false
mos atca-lock-zone config --dry-run=false
mos atca-lock-zone data --dry-run=false
```

Note: This only needs to be done once and once locked, chip configuration cannot be changed anymore. You can dump chip's configuration with `mos atca-get-config --format=yaml`.

Note 2: Sample config is very permissive and is only suitable for development and testing, NOT for production deployments. Please refer to Microchip's manual and other documentation to come up with more secure configuration (we may be able to assist with that too - ask a question on [our forum](#)).

2. Generate a cert and key as normal. An example below shows a self-signed certificate, but of course it doesn't have to be. The important thing is that it's a ECDSA certificate using P256 curve, since that is what the chip supports.

```
openssl ecparam -out ecc.key.pem -name prime256v1 -genkey
openssl req -new -subj \
  "/C=IE/L=Dublin/O=ACME Ltd/OU=Testing/CN=test.acme.com" \
  -sha256 -key ecc.key.pem -text -out ecc.csr tmpl
openssl x509 -in ecc.csr.pem -text -out ecc.crt.pem \
  -req -signkey ecc.key.pem -days 3650
```

3. Write the generated key into the device. Assuming you are using our sample configuration described in the previous section, this is a two-step process:

3.1. Generate and set the key encryption key in slot 4

```
openssl rand -hex 32 > slot4.key
mos -X atca-set-key 4 slot4.key --dry-run=false
AECC508A rev 0x5000 S/N 0x012352aad1bbf378ee, config is locked, data is locked
Slot 4 is a non-ECC private key slot
SetKey successful.
```

3.2. Set the actual ECC key in slot 0

```
mos -X atca-set-key 0 ecc.key.pem --write-key=slot4.key --dry-run=false
AECC508A rev 0x5000 S/N 0x012352aad1bbf378ee, config is locked, data is locked

Slot 0 is a ECC private key slot
Parsed EC PRIVATE KEY
Data zone is locked, will perform encrypted write using slot 4 using slot4.key
SetKey successful.
```

4. Upload the certificate to the device

```
mos put ecc.crt.pem
```

5. Set HTTP server configuration to use the uploaded certificate and private key from device's slot 0:

```
mos config-set http.listen_addr=:443 \
  http.ssl_cert=ecc.crt.pem http.ssl_key=ATCA:0
Getting configuration...
Setting new configuration...
Saving and rebooting...
```

At startup you should see in the device's log:

```
mgos_sys_config_init_http HTTP server started on [443] (SSL)
```

And when connecting with the browser:

```
ATCA:2 ECDH get pubkey ok
ATCA:0 ECDSA sign ok
ATCA:2 ECDH ok
```

Configuring TLS on device's HTTP/WS server

The system HTTP server has the following configuration options (we've added comments to the `mos` tool output):

```
mos config-get http
{
  "enable": true,          # Set to false to disable default HTTP server
  "hidden_files": "",      # Glob pattern for files to hide from serving
  "listen_addr": "80",      # Port to listen on
  "ssl_ca_cert": "",       # CA certificate for mutual TLS authentication
  "ssl_cert": "",          # Certificate file
  "ssl_key": "",           # Private key file
  "upload_acl": "*"        # ACL for which files can be uploaded via /upload
}
```

In order to setup one-way SSL/TLS on the system HTTP server, create a certificate, upload the certificate and the key file to the device, and change HTTP server configuration.

How to create a self-signed certificate

```
openssl req -nodes -new -x509 -keyout key.pem -out cert.pem
mos put cert.pem
mos put key.pem
mos config-set http.listen_addr=443 http.ssl_key=key.pem http.ssl_cert=cert.pem
mos wifi WIFI_NET WIFI_PASS
curl -k https://IP_ADDRESS # Test it !
```

If you want to use mutual (two-way) TLS with the device, follow this procedure to

use a self-signed certificate:

Self-signed certificate for mutual TLS

```
# Common parameters
SUBJ="/C=IE/ST=Dublin/L=Docks/O=MyCompany/CN=howdy"

# Generate CA
openssl genrsa -out ca.key 2048
openssl req -new -x509 -days 365 -key ca.key -out ca.crt \
-subj /C=IE/ST=Dublin/L=Docks/O=mos/CN=me

# Generate client cert
openssl genrsa -out client.key 2048
openssl req -new -key client.key -out client.csr -subj $SUBJ
openssl x509 -req -days 365 -in client.csr -CA ca.crt \
-CAkey ca.key -set_serial 01 -out client.crt

# Generate server cert
openssl genrsa -out server.key 2048
openssl req -new -key server.key -out server.csr -subj $SUBJ
openssl x509 -req -days 365 -in server.csr -CA ca.crt \
-CAkey ca.key -set_serial 01 -out server.crt
```

When done, copy generated files to the device and reconfigure the server:

```
# Upload server key, cert & ca cert to the device
mos put ca.crt
mos put server.key
mos put server.crt

# Update HTTP server settings to use mutual TLS
mos config-set http.ssl_ca_cert=ca.crt http.ssl_cert=server.crt \
http.ssl_key=server.key http.listen_addr=443
```

From that point on, the device should be accessible via secure Websocket:

```
# Configure WiFi
mos wifi WIFI_NET WIFI_PASSWORD
mos --cert-file client.crt \
--key-file client.key \
--port wss://IPADDR/rpc \
call RPC.List
```

Device as a network client

A connected device can be a network client, or network server, or both. For example, if a device provides a RESTful interface, it acts as a network server. If a device connects to an MQTT server, it acts as a network client.

Avoid running a network server on your device for the following reasons:

- It is easy to DoS the device by creating many network connections. Say, a device has 40k of free RAM, and each connection takes 10k, then 4 connections is enough for the denial of service
- A device must implement authentication and authorisation mechanisms that are potentially vulnerable
- A network service code may be vulnerable
- If TLS is used for communication, the connection setup time could be large because of the slow CPU, leading to delays and bad user experience
- TLS certificate management for the local communication could be non-trivial

On the other hand, when a device acts as a client, these problems disappear:

- It is impossible to hack into the device directly because it does not expose any network endpoint
- A device does not care about authentication and authorisation - it is all handled on the cloud side, and secure services like Google IoT Core or AWS IoT would be a good choice for the cloud backend
- The only entity a device should trust is a cloud backend, which is handled by the industry-standard TLS
- No need to keep many network connections, cause a single secure connection to the cloud backend is enough for both management and data flows. This saves precious resources

Securing RPC

RPC is a Mongoose OS mechanism for remote management. Every time `mos` tool is fired in a command line mode or in the GUI mode, RPC is used to query the device - over serial or remotely. Displaying list of files, showing what hardware platform it is, amount of free RAM, toggling GPIO - it is all RPC.

When a new device is flashed, it is in a default state when all RPCs are allowed for everybody. Several mechanisms are available to restrict the access to RPC services:

- Enable authentication - which users are allowed
- Enable authorisation - what allowed users can do
- Disable RPC for the given channel
- Disable all RPC functionality

Enable authentication

Mongoose OS implements authentication using Digest mechanism, the same as [HTTP Digest authentication](#)

employs. It works like this - true for both HTTP Digest and Mongoose OS RPC:

- Client sends a request
- If authentication is not enabled, server replies
- If authentication is enabled and client request has authentication data, server replies
- If authentication is enabled and client request does not have authentication data, server sends back an error requesting authentication with a random nonce to prevent replay attacks
- Client repeats the request with the authentication data created with the server's nonce

In the case of HTTP, an authentication data is sent in the `Authorization`:
HTTP header. In the case of RPC, an authentication data is sent as an additional `auth` key in the RPC frame.

The authentication is enabled by creating a passwords file in the `htdigest` standard format and setting it in the configuration. The format of the passwords file is the same as for the HTTP Digest authentication, and it could be managed by the Apache's `htdigest`.

This is an example that creates a passwords file with user `joe`, uploads that file to the device, and configures RPC to use it:

```
htdigest -c rpc_auth.txt myproduct joe  
mos put rpc_auth.txt  
mos config-set rpc.auth_domain=myproduct  
mos config-set rpc.auth_file=rpc_auth.txt
```

Enable authorisation

RPCs that are authenticated, could be checked against the ACL file to determine whether a given authenticated user can access an RPC:

```
mos put rpc_acl.json  
mos config-set rpc.acl_file=rpc_acl.json
```

The format of the ACL is similar to the config management ACL. Here is an example of `rpc_acl.json` file:

```
[  
 {"method": "FS.*", "acl": "+user1,-user2"},  
 {"method": "*", "acl": "-*"}  
]
```

Disable RPC for given transport

If the transport is disabled, that's it!

RPC transports are serial, HTTP/RESTful,

Websocket, MQTT, Bluetooth. This disables HTTP transport:

```
mos config-set rpc.http.enable=false
```

Disable all RPC functionality

This is the most radical method. Remove `rpc-common` library, and all other `rpc-*` libraries from your `mos.yml`. That totally removes RPC functionality from the firmware.

Note, however, that the RPC security is determined by the security of its transport. For example, RPC over AWS IoT service uses secure, authenticated AWS IoT mechanism, utilising mutual TLS and elaborate policy infrastructure. Thus, encryption, authentication and authorisation of the RPC channel is provided by AWS. Such an RPC channel is highly secure. The opposite example would be an RPC over plain HTTP/RESTful, not authenticated and not locked by authorisation, wide open to the world.

An alternative to RPC for the remote management could be a mechanism provided by the cloud backend - for example, AWS IoT device shadow, or Azure device twin, or Google IoT Core config/state objects, etc.

Device Management Dashboard

Please see [Dashboard Introduction](#).

OTA - Over The Air firmware update

OTA using `mos ota` command

This is the most universal OTA method. It works over any transport: serial connection, BLE, MQTT, Websocket, etc.

```
mos --port DEVICE_ADDRESS ota FIRMWARE_FILENAME_OR_URL
```

OTA using management dashboard file upload

This method is the simplest one. Build, or download a new firmware.

Then, open a management dashboard, select devices you want to update, and upload a .zip file using the Web UI:

The in-progress is visible on the dashboard, regardless of the OTA method:

OTA by calling `OTA.Update` device RPC

```
mos --port DEVICE_ADDRESS ota FIRMWARE_FILENAME_OR_URL
```

OTA using `OTA.Update` RPC call

This method works by downloading a new firmware over HTTP.

NOTE: this method requires professional subscription.

```
mos --port DEVICE_ADDRESS call OTA.Update '{"url": "URL_TO_ZIP"}'
```

OTA using device shadow

This method works by setting a `desired.ota.url` shadow parameter.

As soon as a device comes online, it picks up the change, and downloads the new firmware. NOTE: this method requires professional subscription.

This method works on [mDash](#), AWS IoT, Microsoft Azure.

Just change the device shadow manually or programmatically - here is an example of the mDash online shadow editor:

OTA using HTTP POST

This method is for development. It works only if the device is directly visible. In order to enable HTTP POST OTA handler, include

[ota-http-server](#)

library in your `mos.yml`. Then, you can build a new firmware and push it using this command:

```
curl -v -F file=@build/fw.zip http://IP_ADDR/update
```

OTA architecture overview

OTA implementation in Mongoose OS has 3 layers:

Layer 1. The lowest layer, implemented by the [ota-common](#) library. The API functions that do the work are `updater_context_create()` to start an update, a sequence of `updater_process()` to apply the next chunk of firmware, and `updater_finish()` to finish the update.

Layer 2. The RPC service that provides a remote management API for the low-level OTA API. It is implemented by the [rpc-service-ota](#) library. The RPC API are of two kinds: push and pull.

- `OTA.Update` is a pull mechanism. Works only via HTTP.
You pass a URL to the new .zip file as a parameter of the RPC call, the .zip file gets downloaded and low-level API calls apply it.
- `OTA.{Begin,Write,End}` is a push mechanism. It works over any transport, for example, BLE, or MQTT. They are calling corresponding low-level API.

Layer 3. Helper libraries and tools that invoke the 2nd RPC layer,

- `mos ota` command calls the "push" RPC, and can work over any transport by specifying the `--port ...` parameter. That is the most universal method.
- [ota-http-client](#)
library is able to fetch firmware from a URL. This library is in fact used by the `OTA.Update` RPC implementation. Also, this library is able to periodically poll a given URL for the new version of firmware, and auto-update.
- [ota-http-server](#) library
registers and `/update` URI handler directly on a device. You can push the new firmware over the HTTP POST. This method is for the development purposes: `curl -v -i -F filedata=@fw.zip http://IPADDR/update`
- [ota-shadow](#) library
observes `desired.ota.url` shadow changes. If it gets a new URL, it triggers the `OTA.Update` with that URL. This method works for the offline devices - you can change the shadow of the number of devices, and they will get updated as soon as they come online.

Commit timeout

When a new firmware boots, it can hang or crash for various reasons. It can happen on one of a thousands devices, but still you don't want it to happen on a customer site. The concept of commit timeout is a solution for this problem that makes OTA process reliable - you can push a very untested, dodgy firmware on a production device, and it will roll back if there is any problem.

It works this way: first of, the OTA process never mutates the existing flash partition. OTA-ed firmware and filesystem gets copied to the new partition without touching an existing one.

Secondly, a new firmware boots from a flash partition that is marked "dirty". When a boot loaded sees a "dirty" flag, it starts a hardware timer that triggers after `commit_timeout` seconds. If a "dirty" flag is still present when a timer kicks in, a boot loader regards a firmware bad and reboots back into an old partition.

Thus, a commit timeout feature gives a "verification period" during

which a decision is made whether a new firmware is good or not.

Clearing the dirty flag is done by the `OTA.Commit` call, and could be done either manually or automatically. If the firmware hangs, or crashes before the dirty flag is cleared, the next reboot will roll back.

NOTE: the commit timeout support requires professional subscription.

Implementation details

The main design principle of the OTA mechanisms is reliability: never end up with a bricked device, roll back on any failure. Therefore, an OTA process never updates firmware code or data in-place. Any failure (e.g. power loss) can end up in broken device. Thus, an OTA mechanism uses independent self-contained flash partitions to hold firmware images (code and data), and an intelligent boot loader makes a decision which partition to boot:

Here is a high level overview of the OTA procedure:

1. OTA is triggered via one of the many supported methods:
HTTP POST request,
[periodic timer that polls well known location](#),
[AWS IoT device shadow change](#),
an `OTA.Update` [RPC command](#), or other.
You can create your method using an [OTA API](#).
2. A separate flash partition is created to hold a new firmware image - code and data (root filesystem).
3. A new firmware image is downloaded to the new flash partition. Any failure during that process aborts an OTA.
4. When new firmware image is successfully copied,
 - All files from the old FS that do not exist in the new FS, are moved to the new FS. This is an important mechanism of preserving user data and device-specific configuration, like `conf2.json` - `conf9.json` configuration files, or any other files. Remember: if a firmware image contains a file, it'll override an existing file during OTA.
Never put files like `conf9.json` in your firmware.
 - Boot loader configuration is updated, saying that a new partition exists and the boot loader must boot from it. A new partition is marked dirty, and the "commit interval" time is stored in the boot configuration.
5. Device reboots. Boot loader boots the new partition. It figures out from the boot configuration that that partition is dirty, unsafe, because the "commit" flag is not set. Therefore it starts the hardware timer that will fire after the "commit interval", and executes the new image.
6. The new image starts, performs a usual boot sequence. At some point a `mgos_upd_commit()` is called, which sets a "commit" flag in the boot config, marking this firmware "OK". A commit call could be done automatically after the health-checks, or manually by the user.
If the commit is not made, a boot config still has "commit" flag not set.
7. A boot loader timer handler kicks in. It checks the commit flag. If it is set, it silently exits. If not set, i.e. the firmware is still dirty, the rollback is performed: the image to boot, and commit flag are set to their previous values, and device reboots.

The in-depth example of the OTA on CC3200 is given at

Boot configuration section

If the boot config is stored in only one location, it makes it susceptible to failure during updates, which are usually performed as a read-erase-write operation: a reboot after erase and before write is complete could render device unbootable. The time between the two is short, but we set out to make our update process safe at all points, so we have to deal with it. The way we do it by using two config files with versioning, or sequencing. A sequencer is a monotonically decreasing number, so of the two files the one with smaller sequencer is more recent - on figure 2, config 1 is selected as active because it has smaller sequencer.

When writing a new config file, we always use the currently inactive (older) slot and it will not become newer until it is written - erased config will be older than any valid one because erased NOR flash is filled with all 1s:

mos tool reference

Using --port option

`mos` tool connects to the device specified by `--port` flag, which is set to `auto` by default. That means, `mos` auto-detects the serial port for the device. You can specify this value manually. It could be a serial device, e.g. `--port COM3` on Windows or `--port /dev/ttyUSB0` on Linux.

It is possible to set `--port` value to be a network endpoint instead of serial port. Device listens for commands on serial, Websocket, and MQTT transports (unless they are disabled). Therefore, `--port ws://IP_ADDR/rpc` connects to the remote device via Websocket, and `--port mqtt://MQTT_SERVER/DEVICE_ID/rpc` via the MQTT protocol. That gives an ability to use `mos` tool as a remote device management tool.

Using environment variables to set default option values

The default values for any `mos` flag could be overridden via the environment variable `MOS_FLAGNAME`. For example, to set the default value for `--port` flag, export `MOS_PORT` variable - on Mac/Linux, put that into your `~/.profile`:

```
export MOS_PORT=YOUR_SERIAL_PORT # E.g. /dev/ttyUSB0
```

Boards wiring

In some cases, for example if you're using a bare-bones ESP8266 module instead of a development board, you need to perform extra steps to switch the module between flashing and firmware boot state. This table provides a summary:

Platform	Wiring Notes
----------	--------------

bare bones ESP8266	flash via UART: GPIO15 LOW, GPIO0 LOW, GPIO2 HIGH boot from flash: GPIO15 LOW, GPIO0 HIGH, GPIO2 HIGH boot from SD: GPIO15 HIGH
bare bones ESP32	flash via UART: GPIO0 LOW boot from flash: GPIO0 HIGH
CC3200 launchpad	connect J8 to SOP2 (see guide)

Versioning

The `mos` tool could be self-updated via the Web UI or via the console command `mos update`. The `mos` tool version also influences the firmware build: the libraries that are used during the build correspond to the `mos` version. There are 3 ways you can stay updated:

- Pin to a specific version, e.g. `mos update 1.18`. This is the most stable approach, as nothing gets changed in this case
 - Pin to the "release" channel, `mos update release`. This is the default. Released are created once in 1-2 weeks
 - Pin to the "latest" channel, `mos update latest`. Get the most latest updates, but experience breakages sometimes
- [IL0373 SPIIL0398 SPIIL91874 SPISSD1608 SPISSD1675 SPI](#)
[IL0373 SPIIL0398 SPIIL91874 SPISSD1608 SPISSD1675 SPI](#)

IL0373 SPIIL0398 SPIIL91874 SPISSD1608 SPISSD1675 SPI

Github Repo	C Header	C source	JS source
mongoose-os-libs/arduino-adafruit-epd	Adafruit_EPD.h		api_arduino_adafruit_epd.js

Introduction

This is a port of or rather a wrapper around the [Adafruit EPD Library](#) for [Mongoose OS](#).

Please check [Adafruit EPD Library](#) for additional information and also please note the original [README](#).

Notes:

- this was moved from [bbilger/mgos-arduino-adafruit-epd](#) which was forked from [adafruit/Adafruit_EPD](#)
- EPD stands for e-paper display.

Usage

Examples

There are three examples that show how to use this library:

- [C usage example](#)
- [JavaScript or rather mJS usage example](#)

- C++ usage example

C

```
#include "mgos_aepd.h"
...
// get the e-paper display as configured in mos.yml
Adafruit_EPD *epd = mgos_aepd_get_global();
// methods of the Adafruit_EPD C++ class
// can be invoked through "mgos_aepd_<snake_case_method_name>(epd, ...)"
mgos_aepd_some_method(epd, ...);
...
```

JavaScript / mJS

```
load('api_arduino_adafruit_epd.js');
// get the e-paper display as configured in mos.yml
let epd = ArduinoAdafruitEpd.createGlobal();
// the JavaScript object has the same / most methods the original Adafruit_EPD C++ class has
epd.someMethod(...);
```

C++

```
#include "Adafruit_EPD.h"
#include "mgos_aepd.h"

// get the e-paper display as configured in mos.yml
Adafruit_EPD *epd = mgos_aepd_get_global();
// nothing fancy here, since one can use the original Adafruit_EPD C++ class directly
epd.someMethod(...);
```

Configuration

It's optional to setup the display using the following config but unless you are using more than one display this is probably what you want.

config	type	default	required	comment
aepd.enable	bool	false	-	If enabled, then the display will be setup automatically by the library and the rest of the config must be valid.
aepd.begin	bool	true	-	Calls begin on the display automatically on start.
aepd.driver	string	empty	true	The driver to use for the connected display. At the moment the following drivers are implemented: "IL0373", "IL0398", "IL91874", "SSD1608", "SSD1675".
aepd.width	int	-1	true	Display width in pixels.
aepd.height	int	-1	true	Display height in pixels.
aepd.epd_spi_cs_index	int	-1	true	spi.csX_gpio index for the EDP, 0, 1 or 2.
aepd.sram_spi_cs_index	int	-1	false	spi.csX_gpio index for the SRAM, 0, 1 or 2, or -1 to not use SRAM.
aepd.epd_dc_gpio	int	-1	true	EPD DC GPIO.
aepd.epd_reset_gpio	int	-1	false	EPD reset GPIO or -1
aepd.debug	bool	false	-	At the moment it only prevents a bootloop on misconfiguration.

If you wish to not setup the display via configuration, then you can set it up in C via `mgos_aepd_create`, in C++ by directly instantiating a

concrete subclass of `Adafruit_EPD` (e.g. `Adafruit_SSD1675`), and in JS you are on your own but you can for example "ffi" `mgos_aepd_create`.

License

This library is - like the original one - licensed under [MIT license](#).

writeRAMCommand

```
virtual uint8_t writeRAMCommand(uint8_t index) = 0;
```

setRAMAddress

```
virtual void setRAMAddress(uint16_t x, uint16_t y) = 0;
```

powerUp

```
virtual void powerUp(void) = 0;
```

update

```
virtual void update(void) = 0;
```

powerDown

```
virtual void powerDown(void) = 0;
void hardwareReset(void);
```

JS API

- [Core](#)
- [Cloud services](#)
- [Networking](#)
- [Drivers](#)
- [Arduino compat](#)
- [RPC](#)
- [Misc](#)

JSON

Github Repo	C Header	C source	JS source
cesanta/frozen	frozen.h	frozen.c	

(*json_walk_callback_t)

```
typedef void (*json_walk_callback_t)(void *callback_data, const char *name,
                                     size_t name_len, const char *path,
                                     const struct json_token *token);
```

Callback-based SAX-like API.

Property name and length is given only if it's available: i.e. if current event is an object's property. In other cases, `name` is `NULL`. For example, name is never given:

- For the first value in the JSON string;
- For events `JSON_TYPE_OBJECT_END` and `JSON_TYPE_ARRAY_END`

E.g. for the input `{ "foo": 123, "bar": [1, 2, { "baz": true }] }`,

the sequence of callback invocations will be as follows:

- type: `JSON_TYPE_OBJECT_START`, name: `NULL`, path: "", value: `NULL`
- type: `JSON_TYPE_NUMBER`, name: `"foo"`, path: `".foo"`, value: `"123"`
- type: `JSON_TYPE_ARRAY_START`, name: `"bar"`, path: `".bar"`, value: `NULL`
- type: `JSON_TYPE_NUMBER`, name: `"0"`, path: `".bar[0]"`, value: `"1"`
- type: `JSON_TYPE_NUMBER`, name: `"1"`, path: `".bar[1]"`, value: `"2"`
- type: `JSON_TYPE_OBJECT_START`, name: `"2"`, path: `".bar[2]"`, value: `NULL`
- type: `JSON_TYPE_TRUE`, name: `"baz"`, path: `".bar[2].baz"`, value: `"true"`
- type: `JSON_TYPE_OBJECT_END`, name: `NULL`, path: `".bar[2]"`, value: `{"baz": true}`
- type: `JSON_TYPE_ARRAY_END`, name: `NULL`, path: `".bar"`, value: `[1, 2, {"baz": true }]`
- type: `JSON_TYPE_OBJECT_END`, name: `NULL`, path: "", value: `{"foo": 123, "bar": [1, 2, {"baz": true }] }`

json_walk

```
int json_walk(const char *json_string, int json_string_length,
              json_walk_callback_t callback, void *callback_data);
```

Parse `json_string`, invoking `callback` in a way similar to SAX parsers;
see `json_walk_callback_t`.

Return number of processed bytes, or a negative error code.

json_printf

```
int json_printf(struct json_out *, const char *fmt, ...);
int json_vprintf(struct json_out *, const char *fmt, va_list ap);
```

Generate formatted output into a given sting buffer.

This is a superset of `printf()` function, with extra format specifiers:

- `%B` print json boolean, `true` or `false`. Accepts an `int`.
- `%Q` print quoted escaped string or `null`. Accepts a `const char *`.
- `%.*Q` same as `%Q`, but with length. Accepts `int`, `const char *`
- `%V` print quoted base64-encoded string. Accepts a `const char *, int`.
- `%H` print quoted hex-encoded string. Accepts a `int`, `const char *`.
- `%M` invokes a `json_printf_callback_t` function. That callback function can consume more parameters.

Return number of bytes printed. If the return value is bigger than the supplied buffer, that is an indicator of overflow. In the overflow case, overflowed bytes are not printed.

json_fprintf

```
int json_fprintf(const char *file_name, const char *fmt, ...);
int json_vfprintf(const char *file_name, const char *fmt, va_list ap);
```

Same as json_printf, but prints to a file.

File is created if does not exist. File is truncated if already exists.

json_asprintf

```
char *json_asprintf(const char *fmt, ...);
char *json_vasprintf(const char *fmt, va_list ap);
```

Print JSON into an allocated 0-terminated string.

Return allocated string, or NULL on error.

Example:

```
char *str = json_asprintf("{a:%H}", 3, "abc");
printf("%s\n", str); // Prints "616263"
free(str);
```

json_printf_array

```
int json_printf_array(struct json_out *, va_list *ap);
```

Helper %M callback that prints contiguous C arrays.

Consumes void *array_ptr, size_t array_size, size_t elem_size, char *fmt

Return number of bytes printed.

json_scanf

```
int json_scanf(const char *str, int str_len, const char *fmt, ...);
int json_vscanf(const char *str, int str_len, const char *fmt, va_list ap);
```

Scan JSON string `str`, performing scanf-like conversions according to `fmt`.

This is a `scanf()` - like function, with following differences:

1. Object keys in the format string may be not quoted, e.g. "{key: %d}"
2. Order of keys in an object is irrelevant.
3. Several extra format specifiers are supported:
 - %B: consumes `int *` (or `char *`, if `sizeof(bool) == sizeof(char)`), expects boolean `true` or `false`.
 - %Q: consumes `char **`, expects quoted, JSON-encoded string. Scanned string is malloc-ed, caller must free() the string.
 - %V: consumes `char **`, `int *`. Expects base64-encoded string. Result string is base64-decoded, mallocoed and NUL-terminated. The length of result string is stored in `int *` placeholder. Caller must free() the result.
 - %H: consumes `int *, char **`. Expects a hex-encoded string, e.g. "fa014f". Result string is hex-decoded, mallocoed and NUL-terminated. The length of the result string is stored in `int *` placeholder. Caller must free() the result.
 - %M: consumes custom scanning function pointer and `void *user_data` parameter - see `json_scanner_t` definition.
 - %T: consumes `struct json_token *`, fills it out with matched token.

Return number of elements successfully scanned & converted.

Negative number means scan error.

(*json_scanner_t)

```
typedef void (*json_scanner_t)(const char *str, int len, void *user_data);
```

json_scanf's %M handler

json_scanf_array_elem

```
int json_scanf_array_elem(const char *s, int len, const char *path, int index,
                           struct json_token *token);
```

Helper function to scan array item with given path and index.

Fills `token` with the matched JSON token.

Return -1 if no array element found, otherwise non-negative token length.

json_unescape

```
int json_unescape(const char *src, int slen, char *dst, int dlen);
```

Unescape JSON-encoded string `src`,`slen` into `dst`, `dlen`.

`src` and `dst` may overlap.

If destination buffer is too small (or zero-length), result string is not written but the length is counted nevertheless (similar to `snprintf`).

Return the length of unescaped string in bytes.

json_escape

```
int json_escape(struct json_out *out, const char *str, size_t str_len);
```

Escape a string `str`, `str_len` into the printer `out`.

Return the number of bytes printed.

json_fread

```
char *json_fread(const char *file_name);
```

Read the whole file in memory.

Return malloc-ed file content, or NULL on error. The caller must free().

json_setf

```
int json_setf(const char *s, int len, struct json_out *out,
               const char *json_path, const char *json_fmt, ...);
```

Update given JSON string `s`,`len` by changing the value at given `json_path`.

The result is saved to `out`. If `json_fmt == NULL`, that deletes the key.

If path is not present, missing keys are added. Array path without an index pushes a value to the end of an array.

Return 1 if the string was changed, 0 otherwise.

Example: `s` is a JSON string { "a": 1, "b": [2] }

```
json_setf(s, len, out, ".a", "7"); // { "a": 7, "b": [ 2 ] }
```

```
json_setf(s, len, out, ".b", "7"); // { "a": 1, "b": 7 }
```

```
json_setf(s, len, out, ".b[]", "7"); // { "a": 1, "b": [ 2, 7 ] }
```

```
json_setf(s, len, out, ".b", NULL); // { "a": 1 }
```

json_prettyify

```
int json_prettyify(const char *s, int len, struct json_out *out);
```

Pretty-print JSON string `s`,`len` into `out`.

Return number of processed bytes in `s`.

json_prettyify_file

```
int json_prettyify_file(const char *file_name);
```

Prettyify JSON file `file_name`.

Return number of processed bytes, or negative number of error.

On error, file content is not modified.

json_next_key

```
void *json_next_key(const char *s, int len, void *handle, const char *path,
                    struct json_token *key, struct json_token *val);
```

Iterate over an object at given JSON `path`.

On each iteration, fill the `key` and `val` tokens. It is OK to pass NULL

for `key`, or `val`, in which case they won't be populated.

Return an opaque value suitable for the next iteration, or NULL when done.

Example:

```
void *h = NULL;
struct json_token key, val;
while ((h = json_next_key(s, len, h, ".foo", &key, &val)) != NULL) {
    printf("[%.*s] -> [%.*s]\n", key.len, key.ptr, val.len, val.ptr);
}
```

json_next_elem

```
void *json_next_elem(const char *s, int len, void *handle, const char *path,
                     int *idx, struct json_token *val);
```

Iterate over an array at given JSON `path`.

Similar to `json_next_key`, but fills array index `idx` instead of `key`.

Time

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_time.h	mgos_time.c	

mgos_uptime

```
double mgos_uptime(void);
```

Get number of seconds since last reboot

mgos_uptime_micros

```
int64_t mgos_uptime_millis(void);
```

Get number of milliseconds since last reboot

mgos_time_millis

```
int64_t mgos_time_millis(void);
```

Get wall time in milliseconds.

mgos_strerror

```
int mgos_strerror(char *s, int size, char *fmt, int time);
```

Format `time` according to a `strftime()`-conformant format.

Write the result into the `s,size` buffer. Return resulting string length.

mgos_settimeofday

```
int mgos_settimeofday(double time, struct timezone *tz);
```

Like standard `settimeofday()`, but uses `double` seconds value instead of `struct timeval *tv`. If time was changed successfully, emits an event

`MGOS_EVENT_TIME_CHANGED`.

Net events

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	<code>mgos_net.h</code>	<code>mgos_net.c</code>	<code>api_net.js</code>

Low-level network configuration API.

Contains definitions of the configuration state. Allows to set up an event handler that tracks state changes: when the network connectivity is lost, established, etc.

MGOS_EVENT_BASE

```
#define MGOS_EVENT_GRP_NET MGOS_EVENT_BASE('N', 'E', 'T')
```

Event group which should be given to `mgos_event_add_group_handler()` in order to subscribe to network events.

Example:

```
static void my_net_ev_handler(int ev, void *evd, void *arg) {
    if (ev == MGOS_NET_EV_IP_ACQUIRED) {
        LOG(LL_INFO, ("Just got IP!"));
        // Fetch something very useful from somewhere
    }
    (void) evd;
    (void) arg;
}

// Somewhere else:
mgos_event_add_group_handler(MGOS_EVENT_GRP_NET, my_net_ev_handler, NULL);
```

mgos_net_get_ip_info

```
bool mgos_net_get_ip_info(enum mgos_net_if_type if_type, int if_instance,
                           struct mgos_net_ip_info *ip_info);
```

Retrieve IP configuration of the provided interface type and instance number, and fill provided `ip_info` with it. Returns `true` in case of success, false otherwise.

mgos_net_ip_to_str

```
char *mgos_net_ip_to_str(const struct sockaddr_in *sin, char *out);
```

Converts address to dotted-quad NUL-terminated string.

`out` must be at least 16 bytes long.

Returns the `out` pointer.

mgos_net_str_to_ip

```
bool mgos_net_str_to_ip(const char *ips, struct sockaddr_in *sin);
```

Parses dotted-quad NUL-terminated string into an IPv4 address.

mgos_net_str_to_ip_n

```
bool mgos_net_str_to_ip_n(const struct mg_str ips, struct sockaddr_in *sin);
```

Parses dotted-quad NUL-terminated string into an IPv4 address.

mgos_get_nameserver

```
char *mgos_get_nameserver(void);
```

Returns nameserver address. The caller should `free()` it. Returns NULL in case of no DNS.

JS API

Net.ctos

```
Net.ctos(conn, local, ip, port)
```

Convert address of a connection `conn` to string. Set `local` to `true` to stringify local address, otherwise `false` to stringify remote.

Set ip to true to stringify IP, port to stringify port. Example:

```
print('Connection from:', Net.ctos(conn, false, true, true));
```

Net.discard

```
Net.discard(conn, len)
```

Remove initial `len` bytes of data from the connection's `conn` receive buffer in order to discard that data and reclaim RAM to the system.

Net.serve

```
Net.serve(options)
```

Start TCP or UDP server. `options` is an object:

```
{
  // Required. Port to listen on, 'tcp://PORT' or `udp://PORT` .
  addr: 'tcp://1234',
  // Optional. Called when connection is established.
  onconnect: function(conn) {},
  // Optional. Called when new data is arrived.
  ondata: function(conn, data) {},
  // Optional. Called when protocol-specific event is triggered.
  onevent: function(conn, data, ev, edata) {},
  // Optional. Called when the connection is about to close.
  onclose: function(conn) {},
  // Optional. Called when on connection error.
  onerror: function(conn) {},
}
```

Example - a UDP echo server. Change `udp://` to `tcp://` to turn this example into the TCP echo server:

```
Net.serve({
  addr: 'udp://1234',
  ondata: function(conn, data) {
    print('Received from:', Net.ctos(conn, false, true, true), ':', data);
    Net.send(conn, data);           // Echo received data back
    Net.discard(conn, data.length); // Discard received data
  },
});
```

Net.connect

```
Net.connect(options)
```

Connect to a remote host. `options` is the same as for the `Net.serve`.

The `addr` format is `[PROTO://]HOST:PORT`. `PROTO` could be `tcp` or `udp`. `HOST` could be an IP address or a host name. If `HOST` is a name, it will be resolved asynchronously.

Examples of valid addresses: `google.com:80`, `udp://1.2.3.4:53`,

`10.0.0.1:443`, `[:1]:80`.

Net.close

```
Net.close(conn)
```

Send all pending data to the remote peer,
and disconnect when all data is sent.

Return value: none.

Net.send

```
Net.send(conn, data)
```

Send data to the remote peer. `data` is an mJS string.

Return value: none.

Net.EVENT_GRP

```
Net.EVENT_GRP
```

Net events group, to be used with `Event.addGroupHandler()`. Possible

events are:

- Net.STATUS_DISCONNECTED
- Net.STATUS_CONNECTING
- Net.STATUS_CONNECTED
- Net.STATUS_GOT_IP

UART

GitHub Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_uart.h	mgos_uart.c	api_uart.js

See https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
for more information about UART.

mgos_uart_configure

```
bool mgos_uart_configure(int uart_no, const struct mgos_uart_config *cfg);
```

Apply given UART configuration.

Example:

```
int uart_no = 0;

struct mgos_uart_config ucfg;
mgos_uart_config_set_defaults(uart_no, &ucfg);

ucfg.baud_rate = 115200;
ucfg.rx_buf_size = 1500;
ucfg.tx_buf_size = 1500;

if (!mgos_uart_configure(uart_no, &ucfg)) {
    LOG(LL_ERROR, ("Failed to configure UART%d", uart_no));
}
```

mgos_uart_config_set_defaults

```
void mgos_uart_config_set_defaults(int uart_no, struct mgos_uart_config *cfg);
```

Fill provided `cfg` structure with the default values. See example above.

mgos_uart_config_get

```
bool mgos_uart_config_get(int uart_no, struct mgos_uart_config *cfg);
```

Fill provided `cfg` structure with the current UART config.

Returns false if the specified UART has not been configured yet.

(*mgos_uart_dispatcher_t)

```
typedef void (*mgos_uart_dispatcher_t)(int uart_no, void *arg);
```

UART dispatcher signature, see `mgos_uart_set_dispatcher()`

mgos_uart_set_dispatcher

```
void mgos_uart_set_dispatcher(int uart_no, mgos_uart_dispatcher_t cb,
                             void *arg);
```

Set UART dispatcher: a callback which gets called when there is data in the input buffer or space available in the output buffer.

mgos_uart_write

```
size_t mgos_uart_write(int uart_no, const void *buf, size_t len);
```

Write data to the UART.

Note: if there is enough space in the output buffer, the call will return immediately, otherwise it will wait for buffer to drain.

If you want the call to not block, check `mgos_uart_write_avail()` first.

mgos_uart_write_avail

```
size_t mgos_uart_write_avail(int uart_no);
```

Returns amount of space available in the output buffer.

mgos_uart_printf

```
int mgos_uart_printf(int uart_no, const char *fmt, ...);
```

Write data to UART, printf style.

Note: currently this requires that data is fully rendered in memory before sending. There is no fixed limit as heap allocation is used, but be careful when printing longer strings.

mgos_uart_read

```
size_t mgos_uart_read(int uart_no, void *buf, size_t len);
```

Read data from UART input buffer.

Note: unlike write, read will not block if there are not enough bytes in the input buffer.

mgos_uart_read_mbuf

```
size_t mgos_uart_read_mbuf(int uart_no, struct mbuf *mb, size_t len);
```

Like `mgos_uart_read`, but reads into an mbuf.

mgos_uart_read_avail

```
size_t mgos_uart_read_avail(int uart_no);
```

Returns the number of bytes available for reading.

mgos_uart_set_rx_enabled

```
void mgos_uart_set_rx_enabled(int uart_no, bool enabled);
```

Controls whether UART receiver is enabled.

mgos_uart_is_rx_enabled

```
bool mgos_uart_is_rx_enabled(int uart_no);
```

Returns whether UART receiver is enabled.

mgos_uart_flush

```
void mgos_uart_flush(int uart_no);
```

Flush the UART output buffer - waits for data to be sent.

mgos_uart_schedule_dispatcher

```
void mgos_uart_schedule_dispatcher(int uart_no, bool from_isr);
```

Schedule a call to dispatcher on the next `mongoose_poll`

mgos_uart_get_stats

```
const struct mgos_uart_stats *mgos_uart_get_stats(int uart_no);
```

Get UART statistics

JS API

UART.setConfig

```
UART.setConfig(uartNo, param)
```

Set UART config. `param` is an object with the following optional fields:

- `baudRate`: baud rate, integer, default: 115200;
- `numDataBits`: Number of data bits, default: 8;
- `parity`: Parity: 0 - none, 1 - even, 2 - odd; default: none;
- `numStopBits`: Number of stop bits: 1 - 1 bit, 2 - 2 bits, 3 - 1.5; default: 1;
- `rxBufSize`: size of the Rx buffer, integer, default: 256;
- `rxFlowControl`: whether Rx flow control (RTS pin) is enabled, boolean,

- `rxLingerMicros`: how many microseconds to linger after Rx fifo is empty, in case more data arrives. Integer, default: 15;
- `txBufSize`: size of the Tx buffer, integer, default: 256;
- `txFlowControl`: whether Tx flow control (CTS pin) is enabled, boolean, default: false;

Other than that, there are architecture-dependent settings, grouped in the objects named with the architecture name: "esp32", "esp8266", etc.

Settings for esp32:

```
esp32: {
  /*
   * GPIO pin numbers, default values depend on UART.
   *
   * UART 0: Rx: 3, Tx: 1, CTS: 19, RTS: 22
   * UART 1: Rx: 13, Tx: 14, CTS: 15, RTS: 16
   * UART 2: Rx: 17, Tx: 25, CTS: 26, RTS: 27
   */
  gpio: {
    rx: number,
    tx: number,
    cts: number,
    rts: number,
  },
  /* Hardware FIFO tweaks */
  fifo: {
    /*
     * A number of bytes in the hardware Rx fifo, should be between 1 and 127.
     * How full hardware Rx fifo should be before "rx fifo full" interrupt is
     * fired.
     */
    rxFullThresh: number,
    /*
     * A number of bytes in the hardware Rx fifo, should be more than
     * rx_fifo_full_thresh.
     *
     * How full hardware Rx fifo should be before CTS is deasserted, telling
     * the other side to stop sending data.
     */
    rxFcThresh: number,
    /*
     * Time in uart bit intervals when "rx fifo full" interrupt fires even if
     * it's not full enough
     */
    rxAlarm: number,
    /*
     * A number of bytes in the hardware Tx fifo, should be between 1 and 127.
     * When the number of bytes in Tx buffer becomes less than
     * tx_fifo_empty_thresh, "tx fifo empty" interrupt fires.
     */
    txEmptyThresh: number,
  },
}
```

UART.setDispatcher

```
UART.setDispatcher(uartNo, callback, userdata)
```

Set UART dispatcher

callback which gets invoked when there is a new data in the input buffer

or when the space becomes available on the output buffer.

Callback receives the following arguments: (uartNo, userdata).

UART.write

```
UART.write(uartNo, data)
```

Write data to the buffer. Returns number of bytes written.

Example usage: `UART.write(1, "foobar")`, in this case, 6 bytes will be written.

UART.writeAvail

```
UART.writeAvail(uartNo)
```

Return amount of space available in the output buffer.

UART.read

```
UART.read(uartNo)
```

It never blocks, and returns a string containing
read data (which will be empty if there's no data available).

UART.readAvail

```
UART.readAvail(uartNo)
```

Return amount of data available in the input buffer.

UART.setRxEnabled

```
UART.setRxEnabled(uartNo)
```

Set whether Rx is enabled.

UART.isRxEnabled

```
UART.isRxEnabled(uartNo)
```

Returns whether Rx is enabled.

UART.flush

```
UART.flush(uartNo)
```

Flush the UART output buffer, wait for the data to be sent.

Config

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_sys_config.h	mgos_sys_config.c	api_config.js

A lot of the Mongoose OS functionality is driven by the device configuration.

For example, in order to make a device connected to the MQTT server, there is no need to write a single line of code. It is enough to modify `mqtt.*` configuration settings.

A configuration infrastructure is described in the user guide. Below is the programmatic API for the device configuration.

mgos_sys_config_save

```
bool mgos_sys_config_save(const struct mgos_config *cfg, bool try_once,  
                           char **msg);
```

Save config. Performs diff against defaults and only saves diffs.

Reboot is required to reload the config.

If `try_once` is set, the config will only take effect for one boot.

If return value is false, a message may be provided in `*msg`.

If non-NULL, it must be free()'d.

It is safe to pass a NULL `msg`

mgos_sys_config_save_level

```
bool mgos_sys_config_save_level(const struct mgos_config *cfg,  
                               enum mgos_config_level level, bool try_once,  
                               char **msg);
```

Saves given config at the specified level. Performs diff against level-1.

mgos_sys_config_load_level

```
bool mgos_sys_config_load_level(struct mgos_config *cfg,  
                                enum mgos_config_level level);
```

Loads config up to and including level.

save_cfg

```
bool save_cfg(const struct mgos_config *cfg, char **msg);
```

Deprecated API, equivalent to `mgos_sys_config_save(cfg, false, msg)`.

load_config_defaults

```
bool load_config_defaults(struct mgos_config *cfg);
```

Loads configs up to `MGOS_CONFIG_LEVEL_USER` - 1. Deprecated.

mgos_config_reset

```
void mgos_config_reset(int level);
```

Reset config down to and including `|level|`.

0 - defaults, 1-8 - vendor levels, 9 - user.

`mgos_config_reset(MGOS_CONFIG_LEVEL_USER)` will wipe user settings.

(*mgos_config_validator_fn)

```
typedef bool (*mgos_config_validator_fn)(const struct mgos_config *cfg,
                                         char **msg);
void mgos_sys_config_register_validator(mgos_config_validator_fn fn);
```

Register a config validator.

Validators will be invoked before saving config and if any of them returns false, config will not be saved.

An error message may be *msg may be set to error message.

Note: if non-NULL, *msg will be freed. Remember to use strdup and asprintf.

mgos_config_validate

```
bool mgos_config_validate(const struct mgos_config *cfg, char **msg);
```

Run validators on the specified config.

mgos_expand_mac_address_placeholders

```
void mgos_expand_mac_address_placeholders(char *str);
```

Expands question marks in "str" with digits from the MAC address.

mgos_config_apply

```
bool mgos_config_apply(const char *sys_config_subset_json, bool save);
```

Apply a subset of system configuration. Return true on success.

mgos_config_apply_s

```
bool mgos_config_apply_s(const struct mg_str, bool save);
```

Same as mgos_config_apply but uses mg_str

mgos_sys_config_parse_sub

```
bool mgos_sys_config_parse_sub(const struct mg_str json, const char *section,
                               void *cfg);
```

Parse a subsection of sys config, e.g. just "spi".

cfg must point to the subsection's struct.

Example:

```
struct mgos_config_spi cfg;
const struct mg_str json_cfg = MG_MK_STR("{\"unit_no\": 1}");
memset(&cfg, 0, sizeof(cfg));
mgos_sys_config_parse_sub(json_cfg, "spi", &cfg);
```

JS API

Cfg.get

```
Cfg.get(path)
```

Get the config value by the configuration variable. Currently, only

simple types are returned: strings, ints, booleans, doubles. Objects are not yet supported.

Examples:

```
load('api_config.js');
Cfg.get('device.id');           // returns a string
Cfg.get('debug.level');         // returns an integer
Cfg.get('wifi.sta.enable');     // returns a boolean
```

Cfg.set

```
Cfg.set(obj, opt_save)
```

Set the configuration. `obj` must be a subset of the whole configuration tree. `save` is boolean flag that indicating whether the change should be saved - it could be omitted, in which case it defaults to `true`.

Examples:

```
load('api_config.js');
Cfg.set({wifi: {ap: {enable: false}}}); // Disable WiFi AP mode
Cfg.set({debug: {level: 3}});           // Set debug level to 3
```

mgos_iram.h

GitHub Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_iram.h	mgos_iram.c	

_IRAM_STR_LIT

```
#define _IRAM_STR_LIT(a) #a
#define _IRAM_STR(a) _IRAM_STR_LIT(a)
```

Provides the IRAM macro that creates unique sections under .text so that unused functions can be GC'd.

mgos_file_utils.h

GitHub Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_file_utils.h	mgos_file_utils.c	

mgos_file_copy

```
bool mgos_file_copy(const char *from, const char *to);
```

Copy a file

mgos_file_digest

```
bool mgos_file_digest(const char *fname, mbedtls_md_type_t dt, uint8_t *digest);
```

Compute file's digest. `*digest` must have enough space for the digest type.

mgos_file_copy_if_different

```
bool mgos_file_copy_if_different(const char *from, const char *to);
```

| Copy the file if target does not exist or is different.

GPIO

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_gpio.h	mgos_gpio.c	api_gpio.js

(*mgos_gpio_int_handler_f)

```
typedef void (*mgos_gpio_int_handler_f)(int pin, void *arg);
```

| GPIO interrupt handler signature.

mgos_gpio_set_mode

```
bool mgos_gpio_set_mode(int pin, enum mgos_gpio_mode mode);
```

| Set mode - input or output.

mgos_gpio_set_pull

```
bool mgos_gpio_set_pull(int pin, enum mgos_gpio_pull_type pull);
```

| Set pull-up or pull-down type.

mgos_gpio_setup_input

```
bool mgos_gpio_setup_input(int pin, enum mgos_gpio_pull_type pull);
```

| Sets up a pin as an input and configures pull-up or pull-down.

mgos_gpio_setup_output

```
bool mgos_gpio_setup_output(int pin, bool level);
```

| Sets up pin output while avoiding spurious transitions:
desired output level is configured first, then mode.

mgos_gpio_read

```
bool mgos_gpio_read(int pin);
```

| Read pin input level.

mgos_gpio_write

```
void mgos_gpio_write(int pin, bool level);
```

| Set pin's output level.

mgos_gpio_toggle

```
bool mgos_gpio_toggle(int pin);
```

Flip output pin value. Returns value that was written.

mgos_gpio_read_out

```
bool mgos_gpio_read_out(int pin);
```

Read the value of the output register.

mgos_gpio_set_int_handler

```
bool mgos_gpio_set_int_handler(int pin, enum mgos_gpio_int_mode mode,
                               mgos_gpio_int_handler_f cb, void *arg);
```

Install a GPIO interrupt handler.

This will invoke handler on the main task, which makes it possible to use any functions but may delay servicing of the interrupt. If lower latency is required, use `mgos_gpio_set_int_handler_isr`, but you'll need to understand the implications, which are platform-specific.

Interrupt is automatically cleared once upon triggering. Then it is disabled until the handler gets a chance to run, at which point it is re-enabled. At this point it may re-trigger immediately if the interrupt condition arose again while the handler was pending or running. Handler may use `mgos_gpio_clear_int` to explicitly clear the condition.

Note that this will not enable the interrupt, this must be done explicitly with `mgos_gpio_enable_int()`.

mgos_gpio_set_int_handler_isr

```
bool mgos_gpio_set_int_handler_isr(int pin, enum mgos_gpio_int_mode mode,
                                   mgos_gpio_int_handler_f cb, void *arg);
```

Same as `mgos_gpio_set_int_handler` but invokes handler in ISR context, without the overhead of a context switch. GPIO interrupts are disabled while the handler is running.

mgos_gpio_enable_int

```
bool mgos_gpio_enable_int(int pin);
```

Enable interrupt on the specified pin.

mgos_gpio_disable_int

```
bool mgos_gpio_disable_int(int pin);
```

Disables interrupt (without removing the handler).

mgos_gpio_clear_int

```
void mgos_gpio_clear_int(int pin);
```

Clears a GPIO interrupt flag.

mgos_gpio_remove_int_handler

```
void mgos_gpio_remove_int_handler(int pin, mgos_gpio_int_handler_f *old_cb,  
                                 void **old_arg);
```

Removes a previously set interrupt handler.

If `cb` and `arg` are not NULL, they will contain previous handler and arg.

mgos_gpio_set_button_handler

```
bool mgos_gpio_set_button_handler(int pin, enum mgos_gpio_pull_type pull_type,  
                                  enum mgos_gpio_int_mode int_mode,  
                                  int debounce_ms, mgos_gpio_int_handler_f cb,  
                                  void *arg);
```

Handle a button on the specified pin.

Configures the pin for input with specified pull-up and performs debouncing:
upon first triggering user's callback is invoked immediately but further
interrupts are inhibited for the following debounce_ms milliseconds.

Typically 50 ms of debouncing time is sufficient.

`int_mode` is one of the `MGOS_GPIO_INT_EDGE_*` values and will specify whether
the handler triggers when button is pressed, released or both.

Which is which depends on how the button is wired: if the normal state is
pull-up (typical), then `MGOS_GPIO_INT_EDGE_NEG` is press and
`_POS` is release.

Calling with `cb = NULL` will remove a previously installed handler.

Note: implicitly enables the interrupt.

mgos_gpio_blink

```
bool mgos_gpio_blink(int pin, int on_ms, int off_ms);
```

A utility function that takes care of blinking an LED.

The pin must be configured as output first.

On (output "1") and off ("0") times are specified in milliseconds.

Set to (0, 0) to disable.

mgos_gpio_str

```
const char *mgos_gpio_str(int pin_def, char buf[8]);
```

String representation of pin number.

Will return "PA5" or "PK3" for platforms that have port banks.

JS API

GPIO.set_mode

```
GPIO.set_mode(pin, mode)
```

Set GPIO pin mode.

`mode` can be either `GPIO.MODE_INPUT` or `GPIO.MODE_OUTPUT`.

GPIO.set_pull

```
GPIO.set_pull(pin, pull_type)
```

Set GPIO pin pull type.

`pull_type` can be either `GPIO.PULL_NONE`, `GPIO.PULL_UP`, or `GPIO.PULL_DOWN`.

GPIO.setup_input

```
GPIO.setup_input(pin, pull_type)
```

Setup pin as input and configure pull type.

`pull_type` can be either `GPIO.PULL_NONE`, `GPIO.PULL_UP`, or `GPIO.PULL_DOWN`.

GPIO.setup_output

```
GPIO.setup_output(pin, level)
```

Setup pin as output and set initial level, 0 or 1.

Avoids spurious transitions: applies level first, then sets mode.

GPIO.toggle

```
GPIO.toggle(pin)
```

Toggle the level of certain GPIO pin.

Return value: 0 or 1, indicating the resulting pin level.

GPIO.write

```
GPIO.write(pin, level)
```

Set GPIO pin level to either 0 or 1. Return value: none.

GPIO.read

```
GPIO.read(pin)
```

Read GPIO pin level. Return value: 0 or 1.

GPIO.enable_int

```
GPIO.enable_int(pin)
```

Enable interrupts on GPIO pin.

This function must be called AFTER the interrupt handler is installed.

Return value: 1 in case of success, 0 otherwise.

GPIO.disable_int

```
GPIO.disable_int(pin)
```

Disable interrupts on GPIO pin.

Return value: 1 in case of success, 0 otherwise.

GPIO.blink

```
GPIO.blink(pin, on_ms, off_ms)
```

A utility function that takes care of blinking an LED.

The pin must be configured as output first.

On (output "1") and off ("0") times are specified in milliseconds.

Set to (0, 0) to disable.

Return value: 1 on success, 0 on failure.

GPIO.set_int_handler

```
GPIO.set_int_handler(pin, mode, handler)
```

Install GPIO interrupt handler. `mode` could be one of: `GPIO.INT_NONE`,
`GPIO.INT_EDGE_POS`, `GPIO.INT_EDGE_NEG`, `GPIO.INT_EDGE_ANY`,
`GPIO.INT_LEVEL_HI`, `GPIO.INT_LEVEL_LO`.

Return value: 1 in case of success, 0 otherwise.

Example:

```
GPIO.set_mode(pin, GPIO.MODE_INPUT);
GPIO.set_int_handler(pin, GPIO.INT_EDGE_NEG, function(pin) {
    print('Pin', pin, 'got interrupt');
}, null);
GPIO.enable_int(pin);
```

GPIO.set_button_handler

```
GPIO.set_button_handler(pin, pull, intmode, period, handler)
```

Install

GPIO button handler. `pull` is pull type, `intmode` is interrupt mode,
`period` is debounce interval in milliseconds, `handler` is a function that
receives pin number.

Return value: 1 in case of success, 0 otherwise.

Example:

```
GPIO.set_button_handler(pin, GPIO.PULL_UP, GPIO.INT_EDGE_NEG, 200, function(x) {
    print('Button press, pin: ', x);
}, null);
```

- [App](#)
- [Bitbang](#)
- [Debug \(UART\)](#)
- [Event](#)
- [mgos_file_utils.h](#)
- [GPIO](#)
- [mgos_iram.h](#)
- [Net events](#)
- [Config](#)
- [System](#)
- [Time](#)
- [Timers](#)
- [UART](#)
- [Utils](#)
- [JSON](#)

Utils

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_utils.h	mgos_utils.c	

mgos_system_restart_after

```
void mgos_system_restart_after(int delay_ms);
```

Restart system after the specified number of milliseconds

mgos_rand_range

```
float mgos_rand_range(float from, float to);
```

Return random number in a given range.

Event

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_event.h	mgos_event.c	api_events.js

Mongoose OS provides a way to get a notification when certain event happens. Each event has an associated event data passed as `void *`.

MGOS_EVENT_BASE

```
#define MGOS_EVENT_BASE(a, b, c) ((a) << 24 | (b) << 16 | (c) << 8)
```

Macro to generate unique base event number.

A library can use the last byte (LSB) to create up to 256 unique events (see enum below on how Mongoose OS core defines its events).

A library should call `mgos_event_register_base()` in order to claim it and prevent event number conflicts.

MGOS_EVENT_BASE

```
#define MGOS_EVENT_SYS MGOS_EVENT_BASE('M', 'O', 'S')
```

These events are registered by the MGOS core.

Other events could be registered by the external libraries.

mgos_event_register_base

```
bool mgos_event_register_base(int base_event_number, const char *name);
```

Register a base event number in order to prevent event number conflicts.

Use `MGOS_EVENT_BASE()` macro to get `base_event_number`; `name` is an arbitrary name of the module who registers the base number.

Example:

```
#define MY_EVENT_BASE MGOS_EVENT_BASE('F', '0', '0')

enum my_event {
    MY_EVENT_AAA = MY_EVENT_BASE,
    MY_EVENT_BBB,
    MY_EVENT_CCC,
};

// And somewhere else:
mgos_event_register_base(MY_EVENT_BASE, "my module foo");
```

mgos_event_trigger

```
int mgos_event_trigger(int ev, void *ev_data);
```

Trigger an event `ev` with the event data `ev_data`. Return number of event handlers invoked.

(*mgos_event_handler_t)

```
typedef void (*mgos_event_handler_t)(int ev, void *ev_data, void *userdata);
```

Event handler signature.

mgos_event_add_handler

```
bool mgos_event_add_handler(int ev, mgos_event_handler_t cb, void *userdata);
```

Add an event handler. Return true on success, false on error (e.g. OOM).

Example:

```
static void system_restart_cb(int ev, void *ev_data, void *userdata) {
    LOG(LL_INFO, ("Going to reboot!"));
    (void) ev;
    (void) ev_data;
    (void) userdata;
}

// And somewhere else:
mgos_event_add_handler(MGOS_EVENT_REBOOT, system_restart_cb, NULL);
```

mgos_event_add_group_handler

```
bool mgos_event_add_group_handler(int evgrp, mgos_event_handler_t cb,
                                 void *userdata);
```

Like `mgos_event_add_handler()`, but subscribes on all events in the given group `evgrp`. Event group includes all events from `evgrp & ~0xff` to `evgrp | 0xff`.

Example:

```

static void sys_event_cb(int ev, void *ev_data, void *userdata) {
    LOG(LL_INFO, ("Got system event %d", ev));
    (void) ev;
    (void) ev_data;
    (void) userdata;
}

// And somewhere else:
mgos_event_add_handler(MGOS_EVENT_SYS, sys_event_cb, NULL);

```

mgos_event_remove_handler

```

bool mgos_event_remove_handler(int ev, mgos_event_handler_t cb, void *userdata);
bool mgos_event_remove_group_handler(int evgrp, mgos_event_handler_t cb,
                                     void *userdata);

```

Remove an event handler.

Both cb and userdata must match the initial add invocation.

Returns true if a handler was found and removed, false if there was no such handler in the first place.

JS API

Event.addHandler

```
Event.addHandler(ev, callback, userdata)
```

Add a handler for the given event `ev`. Callback should look like:

```
function(ev, evdata, userdata) { /* ... */ }
```

Example:

```
Event.addHandler(Event.REBOOT, function(ev, evdata, ud) {
    print("Going to reboot!");
}, null);
```

Event.addGroupHandler

```
Event.addGroupHandler(evgrp, callback, userdata)
```

Like `Event.addHandler()`, but subscribes on all events in the given event group `evgrp`. Event group includes all events from `evgrp & ~0xff` to `evgrp | 0xff`.

Example:

```
Event.addGroupHandler(Event.SYS, function(ev, evdata, ud) {
    print("Sys event:", ev);
}, null);
```

Event.on

```
Event.on(event_num, callback, userdata)
```

Alias for `Event.addHandler`

Event.regBase

```
Event.regBase(base_event_number, name)
```

Register a base event number in order to prevent event number conflicts.

Use `Event.baseNumber(id)` to get `base_event_number`; `name` is an arbitrary event name.

Example:

```
let bn = Event.baseNumber("ABC");
if (!Event.regBase(bn, "My module")) {
    die("Failed to register base event number");
}

let MY_EVENT_FOO = bn + 0;
let MY_EVENT_BAR = bn + 1;
let MY_EVENT_BAZ = bn + 2;
```

Event.baseNumber

```
Event.baseNumber(id)
```

Generates unique base event number (32-bit) by a 3-char string.

LSB is always zero, and a library can use it to create up to 256 unique events.

A library should call `Event.regBase()` in order to claim it and prevent event number conflicts. (see example there)

Event.trigger

```
Event.trigger(ev, evdata)
```

Trigger an event with the given id `ev` and event data `evdata`.

Event.evdataLogStr

```
Event.evdataLogStr(evdata)
```

Getter function for the `evdata` given to the event callback for the event

`Event.LOG`, see `Event.addHandler()`.

Event.LOG

```
Event.LOG
```

System event which is triggered every time something is printed to the log. In the callback, use `Event.evdataLogStr(evdata)` to get string which was printed.

Event.REBOOT

```
Event.REBOOT
```

System event which is triggered right before going to reboot. `evdata` is irrelevant for this event.

Event.OTA_STATUS

Event.OTA_STATUS

System event which is triggered when OTA status changes.

In the callback, use `OTA.evdataOtaStatusMsg(evdata)` from `api_ota.js` to get the OTA status message.

Event.CLOUD_CONNECTED

Event.CLOUD_CONNECTED

Triggered when device is connected to the cloud (mqtt, dash)

Event.CLOUD_DISCONNECTED

Event.CLOUD_DISCONNECTED

Triggered when device is disconnected from the cloud

Timers

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_timers.h	mgos_timers.c	api_timer.js

Mongoose OS supports two types of timers: software timers and hardware timers.

- Software timers. Implemented as Mongoose library events, in software.
Timer callback is called in a Mongoose task context. Frequency is specified in milliseconds. Number of software timers is not limited.
Timer intervals cannot be short - limited by the underlying task scheduling. For example, if you want a very frequent sensor reading, like thousand readings a second, use hardware timer instead.
Both C and JavaScript API is available.
- Hardware timers. Implemented in hardware. Timer callback is executed in the ISR context, therefore it can do a limited set of actions.
Number of hardware timers is limited: (ESP8266: 1, ESP32: 4, CC32xx: 4).
Frequency is specified in microseconds. Only C API is present, because calling to JS requires switching to Mongoose task context.

Usage example:

```

#include "mgos_app.h"
#include "mgos_system.h"
#include "mgos_timers.h"

static void my_timer_cb(void *arg) {
    bool val = mgos_gpio_toggle(mgos_sys_config_get_pins_led());
    LOG(LL_INFO, ("uptime: %.2lf", mgos_uptime()));
    (void) arg;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_set_timer(1000, MGOS_TIMER_REPEAT, my_timer_cb, NULL);
    return MGOS_APP_INIT_SUCCESS;
}

```

(*timer_callback)

```
typedef void (*timer_callback)(void *param);
```

Timer callback

mgos_set_timer

```
mgos_timer_id mgos_set_timer(int msecs, int flags, timer_callback cb,
                             void *cb_arg);
```

Setup a timer with `msecs` timeout and `cb` as a callback.

`flags` is a bitmask, currently there's only one flag available:

`MGOS_TIMER_REPEAT` (see above). `arg` is a parameter to pass to `cb`.

Return numeric timer ID.

Note that this is a software timer, with fairly low accuracy and high jitter.

However, number of software timers is not limited.

If you need intervals < 10ms, use `mgos_set_hw_timer`.

Example:

```

static void my_timer_cb(void *arg) {
    bool val = mgos_gpio_toggle(mgos_sys_config_get_pins_led());
    LOG(LL_INFO, ("uptime: %.2lf", mgos_uptime()));
    (void) arg;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_set_timer(1000, MGOS_TIMER_REPEAT, my_timer_cb, NULL);
    return MGOS_APP_INIT_SUCCESS;
}

```

mgos_set_hw_timer

```
mgos_timer_id mgos_set_hw_timer(int usecs, int flags, timer_callback cb,
                                void *cb_arg);
```

Setup a hardware timer with `usecs` timeout and `cb` as a callback.

This is similar to `mgos_set_timer`, but can be used for shorter intervals
(note that time unit is microseconds).

Number of hardware timers is limited (ESP8266: 1, ESP32: 4, CC32xx: 4).

Callback is executed in ISR context, with all the implications of that.

mgos_clear_timer

```
void mgos_clear_timer(mgos_timer_id id);
```

Disable timer with a given timer ID.

mgos_uptime

```
double mgos_uptime(void);
```

Get number of seconds since last reboot

mgos_strerror

```
int mgos_strerror(char *s, int size, char *fmt, int time);
```

Format `time` according to a `strftime()`-conformant format.

Write the result into the `s,size` buffer. Return resulting string length.

JS API

Timer.set

```
Timer.set(milliseconds, flags, handler, userdata)
```

Setup timer with `milliseconds` timeout and `handler` as a callback.

`flags` can be either 0 or `Timer.REPEAT`. In the latter case, the call will be repeated indefinitely (but can be cancelled with `Timer.del()`), otherwise it's a one-off.

Return value: numeric timer ID.

Example:

```
// Call every second
Timer.set(1000, Timer.REPEAT, function() {
  let value = GPIO.toggle(2);
  print(value ? 'Tick' : 'Tock');
}, null);
```

Timer.now

```
Timer.now()
```

Return current time as double value, UNIX epoch (seconds since 1970).

Timer.del

```
Timer.del(id)
```

Cancel previously installed timer.

Timer.fmt

```
Timer(fmt, time)
```

Formats the time 'time' according to the strftime-like format specification 'fmt'. The strftime reference can be found e.g. [here](#).

Example:

```
let now = Timer.now();
let s = Timer fmt("Now it's %I:%M%p.", now);
print(s); // Example output: "Now it's 12:01AM."
```

App

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_app.h	mgos_app.c	

Definitions for the user-defined app entry point, `mgos_app_init()`.

The `mgos_app_init()` function is like the `main()` function in the C program. This is a app's entry point.

The mongoose-os core code does implement `mgos_app_init()` stub function as a weak symbol, so if user app does not define its own `mgos_app_init()`, a default stub will be used. That's what most of the JavaScript based apps do - they do not contain C code at all.

mgos_app_init

```
enum mgos_app_init_result mgos_app_init(void);
```

User app init function.

A weak stub is provided in `mgos_app_init.c`, which can be overridden.

Example of a user-defined init function:

```
#include "mgos_app.h"

enum mgos_app_init_result mgos_app_init(void) {
    if (!my_super_duper.hardware_init()) {
        LOG(LL_ERROR, ("something went bad"));
        return MGOS_APP_INIT_ERROR;
    }
    LOG(LL_INFO, ("my app initialised"));
    return MGOS_APP_INIT_SUCCESS;
}
```

mgos_app_preinit

```
void mgos_app_preinit(void);
```

An early init hook, for apps that want to take control early in the init process. How early? very, very early. If the platform uses RTOS, it is not running yet. Dynamic memory allocation is not safe. Networking is not running. The only safe thing to do is to

communicate to mg_app_init something via global variables or shut down the processor and go (back) to sleep.

Debug (UART)

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_debug.h	mgos_debug.c	api_log.js

mgos_debug_write

```
void mgos_debug_write(int fd, const void *buf, size_t len);
```

Write debug info `buf`, `len` to the given file descriptor `fd`.

mgos_debug_flush

```
void mgos_debug_flush(void);
```

Flush debug UARTs, both stdout and stderr.

mgos_set_stdout_uart

```
bool mgos_set_stdout_uart(int uart_no);
```

Set UART for stdout. Negative value disables stdout.

mgos_set_stderr_uart

```
bool mgos_set_stderr_uart(int uart_no);
```

Set UART for stderr. Negative value disables stderr.

mgos_get_stdout_uart

```
int mgos_get_stdout_uart(void);
```

Get stdout UART number; -1 indicates that stdout is disabled.

mgos_get_stderr_uart

```
int mgos_get_stderr_uart(void);
```

Get stderr UART number; -1 indicates that stderr is disabled.

mgos_debug_suspend_uart

```
void mgos_debug_suspend_uart(void);
```

Suspend UART output (both stdout and stderr); see `mgos_debug_resume_uart()`. Nested suspension is supported: UART needs to be resumed as many times as it was suspended.

mgos_debug_resume_uart

```
void mgos_debug_resume_uart(void);
```

Resume previously suspended UART output, see `mgos_debug_suspend_uart()`.

mgos_debug_uart_is_suspended

```
bool mgos_debug_uart_is_suspended(void);
```

Returns whether UART output is suspended at the moment.

JS API

Log.print

```
Log.print(level, msg)
```

Print message to stderr if provided

level is `>= Cfg.get('debug.level')`. Possible levels are:

- `Log.ERROR` (0)
- `Log.WARN` (1)
- `Log.INFO` (2)
- `Log.DEBUG` (3)
- `Log.VERBOSE_DEBUG` (4)

Log.error

```
Log.error(msg)
```

Shortcut for `Log.print(Log.ERROR, msg)`

Log.warn

```
Log.warn(msg)
```

Shortcut for `Log.print(Log.WARN, msg)`

Log.info

```
Log.info(msg)
```

Shortcut for `Log.print(Log.INFO, msg)`

Log.debug

```
Log.debug(msg)
```

Shortcut for `Log.print(Log.DEBUG, msg)`

Log.verboseDebug

```
Log.verboseDebug(msg)
```

Shortcut for `Log.print(Log.VERBOSE_DEBUG, msg)`

Bitbang

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_bitbang.h	mgos_bitbang.c	api_bitbang.js

Bit banging is a technique to implement hardware protocols fully in software, by managing microcontroller pin states - including timing, voltage levels, etc.

See [wikipedia article](#) for a detailed information.

Usage example:

```
#include "mgos_bitbang.h"
#include "mgos_gpio.h"
#include "mgos_system.h"

void mgos_neopixel_show(struct mgos_neopixel *np) {
    mgos_gpio_write(np->pin, 0);
    mgos_usleep(60);
    mgos_bitbang_write_bits(np->pin, MGOS_DELAY_100NSEC, 3, 8, 8, 6, np->data,
                           np->num_pixels * NUM_CHANNELS);
    mgos_gpio_write(np->pin, 0);
    mgos_usleep(60);
    mgos_gpio_write(np->pin, 1);
}
```

mgos_bitbang_write_bits

```
void mgos_bitbang_write_bits(int gpio, enum mgos_delay_unit delay_unit, int t0h,
                             int t0l, int t1h, int t1l, const uint8_t *data,
                             size_t len);
```

Bit bang GPIO pin `gpio`. `len` bytes from `data` are sent to the specified pin bit by bit. Sending each bit consists of a "high" and "low" phases, length of which is determined by the specified timing parameters.

```
+---+
|   |
-+   +---+
tXh  tXI
```

`t0h` and `t0l` specify timings if the bit being transmitted is 0, `t1h` and `t1l` specify the same for the case where the bit is 1. If any of these is < 0, the corresponding phase is skipped.

mgos_bitbang_write_bits_js

```
void mgos_bitbang_write_bits_js(int gpio, enum mgos_delay_unit delay_unit,
                               uint32_t t, const uint8_t *data, size_t len);
```

This function is a wrapper for `mgos_bitbang_write_bits()`. It has smaller number of arguments (less than 6) and therefore could be FFI-ed to JavaScript. Essentially, it just packs all time patterns into a single value `t`.

JS API

BitBang.write

```
BitBang.write(pin, delay_unit, t0h, t0l, t1h, t1l, ptr, len)
```

Write bits to a given pin. `delay_unit` is one of the:

`BitBang.DELAY_MSEC`, `BitBang.DELAY_USEC`, `BitBang.DELAY_100NSEC`.

`ptr`, `len` is a bit pattern to write. `t0h`, `t0l` is the time pattern

for zero bit, `t1h`, `t1l` is the time pattern for 1. The time pattern

specifies the number of time units to hold the pin high, and the number

of units to hold the pin low. Return value: none.

System

Github Repo	C Header	C source	JS source
cesanta/mongoose-os	mgos_system.h	mgos_system.c	api_sys.js

HAL that needs to be implemented for each hardware platform.

mgos_get_heap_size

```
size_t mgos_get_heap_size(void);
```

Get system memory size.

mgos_get_free_heap_size

```
size_t mgos_get_free_heap_size(void);
```

Get system free memory.

mgos_get_min_free_heap_size

```
size_t mgos_get_min_free_heap_size(void);
```

Get minimal watermark of the system free memory.

mgos_get_fs_memory_usage

```
size_t mgos_get_fs_memory_usage(void);
```

Get filesystem memory usage

mgos_get_fs_size

```
size_t mgos_get_fs_size(void);
```

Get filesystem size.

Deprecated, please use `mgos_vfs_get_space_total("/")` instead.

mgos_get_free_fs_size

```
size_t mgos_get_free_fs_size(void);
```

Get filesystem free space.

Deprecated, please use mgos_vfs_get_space_free("/") instead.

mgos_fs_gc

```
void mgos_fs_gc(void);
```

Garbage-collect filesystem

mgos_wdt_feed

```
void mgos_wdt_feed(void);
```

Feed watchdog

mgos_wdt_set_timeout

```
void mgos_wdt_set_timeout(int secs);
```

Set watchdog timeout

mgos_wdt_enable

```
void mgos_wdt_enable(void);
```

Enable watchdog

mgos_wdt_disable

```
void mgos_wdt_disable(void);
```

Disable watchdog

mgos_system_restart

```
void mgos_system_restart(void) __attribute__((noreturn));
```

Restart system

mgos_msleep

```
void mgos_msleep(uint32_t msecs);
```

Delay given number of milliseconds

mgos_usleep

```
void mgos_usleep(uint32_t usecs);
```

Delay given number of microseconds

mgos_ints_disable

```
void mgos_ints_disable(void);
```

Disable interrupts

mgos_ints_enable

```
void mgos_ints_enable(void);
```

Enable interrupts

(*mgos_cb_t)

```
typedef void (*mgos_cb_t)(void *arg);
```

Callback for `mgos_invoke_cb()`

mgos_invoke_cb

```
bool mgos_invoke_cb(mgos_cb_t cb, void *arg, bool from_isr);
```

Invoke a callback in the main MGOS event loop.

Returns true if the callback has been scheduled for execution.

mgos_get_cpu_freq

```
uint32_t mgos_get_cpu_freq(void);
```

Get the CPU frequency in Hz

mgos_rlock_create

```
struct mgos_rlock_type *mgos_rlock_create(void);
```

Create a recursive lock.

mgos_rlock

```
void mgos_rlock(struct mgos_rlock_type *l);
```

Acquire a recursive lock.

mgos_runlock

```
void mgos_runlock(struct mgos_rlock_type *l);
```

Release a recursive lock.

mgos_rlock_destroy

```
void mgos_rlock_destroy(struct mgos_rlock_type *l);
```

Destroy a recursive lock.

JS API

Sys._sbuf

```
Sys._sbuf(len)
```

Helper function to allocate string of at least given length. Note that the resulting string is usually bigger than this, and it is always longer than 5 bytes; that's to guarantee that the string data is stored in a common buffer and not inlined into mjs_val_t, thus the buffer can be used as an "output" buffer: a string can be passed to some function which will alter the contents, and these changes will be visible to the caller.

Sys.calloc

```
Sys.calloc(nmemb, size)
```

Allocate a memory region.

Note: currently memory allocated this way must be explicitly released with free().

Sys.total_ram

```
Sys.total_ram()
```

Return total available RAM in bytes.

Sys.free_ram

```
Sys.free_ram()
```

Return free available RAM in bytes.

Sys.reboot

```
Sys.reboot(ms)
```

Reboot the system after ms milliseconds. Return value: none.

Sys.uptime

```
Sys.uptime()
```

Return number of seconds since last reboot.

Sys.usleep

```
Sys.usleep(microseconds)
```

Sleep given number of microseconds.

Return value: none.

Sys.wdt_feed

```
Sys.wdt_feed()
```

Feed the watchdog timer.

Return value: none.

CCM (Cloud Communication Module)

See <https://mdash.net/docs/quickstart/ccm.md>

Licensing

- Login to the [Mongoose License Manager](#)
- Buy any number of "ccm" licenses you need. IMPORTANT: choose "ccm" license type:
- If your device is connected to your workstation over USB, run:

```
mos license
```

- Or, if your device is accessible remotely, run:

```
mos --port DEVICE_ADDRESS license
```

Licenses are bound to devices, and they are permanent.

The `mos license` command creates a unique license string specific for a device, and updates `device.license` configuration parameter, which "unlocks" the device and removes restrictions. The License Manager decreases the number of available licenses and saves a unique license string. The next time `mos license` is run for the same device, an old license string is returned by the License Manager but the number of available licenses is not decreased. Thus a device, once licensed, is licensed permanently.

- [User guide](#)
- [Licensing](#)

REST API Reference

Every API call must be authenticated by sending `Authorization: Bearer KEY` HTTP header. Those API calls that use `POST` or `PUT`, should specify `application/json` mime type. Example - calling device's RPC function:

```
curl -H 'Content-Type: application/json' \
-H 'Authorization: Bearer API_KEY' \
-d '{"pin": 2}' \
https://dash.mongoose-os.com/api/v2/devices/DEVICE_ID/rpc/GPIO.Toggle
true
```

GET /devices

List all registered devices. Params: none.

POST /devices

Register new device. Params: none.

POST /devices/:id

Change device properties. Params:

```
{  
  "name": "MyCoolDeviceName",  
  "shared_with": "github_user1,github_user2",  
  "shadow": ...  
}
```

Any key in the params is optional.

DELETE /devices/:id

Delete device. Params: none.

POST /devices/:id/rpc/:func

Call device's RPC function. Params: any valid JSON string, which is expected by the function.

POST /devices/:id/ota

Perform device OTA. Params: a binary content of the firmware .zip file. Example: curl -v -F file=@fw.zip URL.

GET /devices/:id/data

Get device saved data. Params: none.

GET /keys

List all API keys. Params: none.

POST /keys

Create an API key. Params: none.

DELETE /keys/:id

Delete an API key. Params: none.

POST /logs

Get stored notification logs. Params: {"start": 0} - an offset to start from.

GET /devices/data

Get data from all devices. Params: none.

Device Management Dashboard



Mongoose OS provides an integrated service for device management called mDash. It is located at <https://dash.mongoosesh.com/>, and provides the following functionality:

- 24/7 access and monitoring for your devices in the field
- Full isolation. Other users cannot see traffic from your devices and cannot access them
- OTA updates, on-the-fly file editing, config editing, shadow editing, calling any device function like `Sys.Reboot`, `Sys.GetInfo`, `GPIO.Toggle`, etc
- RESTful API and Web UI that provide:
 - Online/offline status together with device metadata like firmware version, build time, device architecture
 - Full access to the devices' RPC services (remote management)
 - Reliable OTA updates with manual or automatic commit
 - Device twin (or, shadow) cloud object, semantically identical to AWS / Azure
- Notification stream that lets you catch state changes, implement custom logging, etc

Device registration

Step 1. Complete steps 1-7 [quickstart guide](#)

Step 2. Login to the <https://dash.mongoosesh.com/>, register a new device.

Left Click on a "Token" link to copy it to the clipboard.

Step 3. Configure device:

```
mos config-set dash.enable=true dash.token=TOKEN
```

Done! Now your device should appear "green" (online) on a dashboard.

Troubleshooting

If `mos config-set` command on Step 5 fails for you, complaining about non-existing variables, then your app needs to be rebuilt with the `dash` library. Edit your app's `mos.yml`:

```
libs:  
...  
- origin: https://github.com/mongoosesh/mongoosesh-lib-dash # <-- Add this line!
```

Then, `mos build`, `mos flash`, and repeat registration from Step 3.

Using device simulator

If you don't want to use a real device for some reason, there is a POSIX program for Linux/Mac/Windows that could be used to simulate the device. Clone it from <https://github.com/cesanta/mongoosesh-device-simulator>.

Then simply run `make`. You'll see the access token prompt:

```
$ make
cc -W -Wall -g -O2 -D MG_ENABLE_CALLBACK_USERDATA=1 -D MG_ENABLE_SSL -DMG_SSL_IF=MG_SSL_IFMBEDTLS -Imbedtls -Imbedcrypto -Imbedx509 main
./simulator
Enter access token:
```

Login to dash.mongooseshosting.com, choose some registered device, copy the access token to the clipboard, paste into the prompt and press enter.

Licensing and Contact

mDash is distributed by Mongoose OS team as a standalone docker image that could be run on-premises, as well as a hosted service <https://dash.mongooseshosting.com>.

In order to run mDash on-premises, install [Docker](#) and execute `docker run mgos/dash`. Customisation can be done via the configuration. By default, there is a restriction on the maximum number of users (5 maximum), with maximum of 2 devices per user.

Please [contact us](#) if you would like to use the mDash in your production setup with increased limits.

- [Introduction](#)
- [Device registration](#)
- [Technical overview](#)
- [OTA support](#)
- [Device shadow](#)
- [Notifications](#)
- [Device sharing](#)
- [Mobile app](#)
- [REST API reference](#)

Technical overview

A configured device connects to the dashboard over the secure Websocket, sending a `Authorization: Bearer GENERATED_TOKEN` handshake header. After the successful handshake, all communication is done via the JSON-RPC 2.0 protocol. Each RPC frame wrapped into the Websocket frame.

The RESTful API endpoints that the dashboard exports start with `/api/v2/`. The endpoints that are for the individual device start with `/api/v2/devices/:id`, where `:id` is the unique device ID, generated at the device registration together with the access token.

The dashboard gives a RESTful access to all RPC services exported by the device. Thus, the dashboard acts as a JSON-RPC / RESTful bridge. The device's RPC methods are mapped to `/api/v2/devices/:id/rpc/:method` endpoints. If the RPC endpoint does not accept any parameters, you can use `GET` HTTP method. If it does, `POST` must be used. The parameters should be a JSON string, and the `Content-Type: application/json` header must be set, e.g.:

```
curl \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer API_KEY' \
-d '{"pin": 2}' \
http://dash.mongoos-e.com/api/v2/devices/DEVICE_ID/rpc/GPIO.Toggle
```

When a device comes online, the `dash` library that is responsible for the dashboard connection, sends a shadow update message with the information about the device: architecture, firmware version, etc:

Notifications

The dashboard provides a special secure Websocket endpoint

`wss://dash.mongoos-e.com/api/v2/notify`. This is a read-only notifications endpoint.

Each notification is a JSON object with three keys:

- `name`: notification name, e.g. "online", "offline", "rpc.in.GetInfo", "rpc.out.Log"
- `id`: an ID of a device that generated the event
- `data`: optional notification-specific data.

The `online` and `offline` events are generated by the dashboard. The `rpc.out.*` events are generated by the device: these are JSON-RPC requests without an ID (notifications). For example, `dash` library forwards all device logs to the dashboard as `Log` RPC calls, thus generating `rpc.out.Log` events. RPC call to the device generate `rpc.in.*` events.

The dashboard UI uses `/api/v2/notify` endpoint in order to catch state changes.

Login to the dashboard and open the developer tools / network / WS panel to see it in action.

How to catch notifications from all devices

You can implement your own service that attaches to the `/api/v2/notify`, for example in Node JS (don't forget to substitute `API_TOKEN` with your real API access token):

```
const Websocket = require('ws');
const addr = 'wss://dash.mongoos-e.com/api/v2/notify?access_token=API_TOKEN';
const ws = new Websocket(addr, { origin: addr });
ws.on('message', msg => console.log('Got message:', msg.toString()));
```

To see your mDash API tokens, login to mDash and click on "Keys" tab:

How to send a notification from your device

The API is documented at [mDash API](#).

Below are quick examples.

In C/C++:

```
mgos_dash_notifyf("MyStat", "{temperature: %f}", 12.34);
```

In JavaScript:

```
Dash.notify('MyStat', {temperature: 12.34});
```

Note that the format of data is arbitrary - it could be anything: a number, a string, an array, or a complex nested object. It's up to you. In this example, a simple object {"temperature": 12.34} is used. The example generates the following notification:

```
{
  "id": "DEVICE_ID",           // This is the device ID that generated notification
  "name": "rpc.out.MyStat",    // "rpc.out.EVENT_NAME", in our case rpc.out.MyStat
  "data": {"temperature": 12.34} // Your event payload - arbitrary
}
```

Further possible actions:

- Catch this notification (see section above) and do your custom action
- Modify the PWA mobile app, which catches all notifications, and show the data somehow to the user
- Save the data automatically to the database, and graph it. See [Data Storage](#) section

Device sharing

It is possible to share a device with other users. This is driven by the `shared_with` property of the device, which is a comma-separated string of user IDs to share device with. The `shared_with` property could be set via the API, or via the dashboard's "Info" panel:

Shared devices are marked with a user icon in the device list:

In order to see your user ID, click on the Account tab: # Mobile app

For every registered user, mDash serves custom Progressive Web App (PWA).

This gives an ability to manage user's devices via a mobile phone.

In order to enable PWA,

- Choose "Mobile App" in the menu
- Tick the "Enable mobile app" checkbox
- Enter your email address and press on "Send invitation" button

That creates a built-in mobile app for the whole fleet of devices.

Customizing built-in app

The default app is customisable. It displays a list of devices shared with the particular mobile app customer, and for each device, displays a list of control widgets. The information displayed by each widget is based on a device shadow. Read-only widgets just display certain shadow values, and "actuator" widgets (switches, buttons) modify shadow values. All device changes are caught via the websocket notifications, and displayed real-time.

The look and feel of the built-in app could be customised in by changing the `var ui = { ... };` object in the PWA code. This is the default value:

```
var ui = {
  title: 'My devices',
  // icon: 'fa-save',
  controls: [
    { path: 'on', type: 'toggle', title: 'LED on/off:' },
    { path: 'uptime', type: 'value', format: 'up: %D', icon: 'fa-clock-o' },
  ],
};
```

As you see, you could modify app's title, menu icon, and define an arbitrary number of control widgets. The description of the available widgets is below.

Toggle button

```
{ path: 'SHADOW_KEY', type: 'toggle', title: 'YOUR TITLE' }
```

Renders a toggle button with a label. A shadow key points to a boolean true/false.

Example:

```
var ui = {
  title: 'Cool app',
  controls: [
    { path: 'on', type: 'toggle', title: 'my light:' },
  ],
};
```

Value

```
{ path: 'SHADOW_KEY', type: 'value', icon: 'fa-save', format: '', title: 'hello' }
```

Renders shadow value. icon is optional, could be a font-awesome class, e.g. fa-save, or an inline base64-encoded icon data. The format could be:

- %D expects a number of seconds, converts it to a Xd Xh Xm Xs.
- %.Nf formats floating-point value, e.g. %.1f leaves one digits after comma.

Example:

```
var ui = {
  title: 'Cool app',
  controls: [
    { path: 'on', type: 'toggle', title: 'my light:' },
    { path: 'uptime', type: 'value', title: 'up:', icon: 'fa-clock-o', format: 'up: %D' },
  ],
};
```

Push button

```
{ path: 'SHADOW_KEY', type: 'button', icon: 'fa-save', title: 'hello' }
```

Renders a push button. icon is optional, could be a font-awesome class, e.g. fa-save, or an inline base64-encoded icon data. The shadow value must be an numeric value. Pushing a button will increment it and set incremented value in desired.

Example:

```

var ui = {
  title: 'Cool app',
  controls: [
    { path: 'on', type: 'toggle', title: 'my light:' },
    { path: 'uptime', type: 'value', title: 'up:', icon: 'fa-clock-o', format: 'up: %D' },
    { path: 'v1', type: 'button', title: 'start action', icon: 'fa-flash' },
  ],
};

```

Fully custom app

You can save your own code and totally modify the look and feel of the app.

If you do so, you can always restore the built-in app by pressing the

"Restore default app" button.

Mobile app API

mDash exports a different set of RESTful API to the mobile app. Mobile app

API allows to get a list of devices, and render the UI according to their shadows.

Method	Endpoint	Params	Description
GET	/api/v2/appuser		Get list of devices
POST	/api/v2/appuser/device/:id	{shadow}	Update device shadow
WSS	/api/v2/appuser/ws		WebSocket notification URI

Device shadow

Dashboard implements device shadow mechanism, semantically identical to
to the Azure device twin and Amazon IoT device shadow. Google IoT Core
state/config objects provide similar functionality, but implemented quite differently.

Device shadow is a JSON object that is associated with a device. It
lives on a cloud, keeps device state, and always available
regardless whether the associated device is online or offline. Shadow object
has four top-level keys:

- `desired` - this is the state you want your device to have.
- `reported` - this is the state your device actually has.
- `delta` - automatically generated by the cloud
every time `desired` or `reported` changes.
`delta` is the difference between `desired` and `reported`
only for keys that are present in `desired`.
- `tags` - arbitrary settings invisible to the device but visible to
the cloud user.

The structure of the `desired` and `reported` subobjects is arbitrary -
create whatever structure you wish. However, the `dash` library that connects
your device to the dashboard, reserves some keys:

- `reported.ota` - object that keeps firmware information and last OTA status
- `reported.stats` - object that keeps device RAM statistics

The device shadow is displayed in the device list, and it is available for

edit in the device panel (when clicked on the device name):

Devices can update their shadow using C API or JavaScript API, see [shadow library](#) for reference.

In order to create new keys, send a shadow update with that new key and its value. In order to delete a key, send a shadow update where that key is set to `null`.

The best practice for using shadow on the device side is this:

- Catch the `MGOS_SHADOW_CONNECTED` event that indicates that the device is connected to its shadow, and report device current state to the `reported` key.
- Catch `MGOS_SHADOW_UPDATE_DELTA` event, and apply all keys in the `delta` to the device. If the state is changed, report and update to the `reported` key.

Example: see [example-shadow-js](#) app for the JavaScript example, and <https://github.com/cesanta/mongoose-os-smart-light> for C example.

Both implement the canonical best practice mentioned above.

OTA support



If a device includes `rpc-service-ota` library, then it can be updated remotely.

The three RPC functions that perform the OTA are `OTA.Begin`, `OTA.Write` and `OTA.End`.

mDash provides a convenience RESTful handler for the OTA, where you can just `POST` the new firmware .zip file, and mDash will call `OTA.Begin` followed by a sequence of `OTA.Write` calls, finished by `OTA.End`.

The OTA can be performed either via the Web UI, or programmatically using the REST API:

```
curl -H 'Authorization: Bearer API_KEY' \
-v -F file=@fw.zip
http://dash.mongoose-os.com/api/v2/devices/DEVICE_ID/ota
true
```

Once the firmware is updated, the device reboots in the "dirty", uncommitted state. An `OTA.Commit` call must be done to bless the new firmware, otherwise it will rollback, thinking that the health check did not pass. You can call `OTA.Commit` as any other RPC method. mDash provides a handy commit button for the convenience, when it sees an uncommitted device.

Data logging and visualisation

mDash provides an experimental support for data logging

and visualisation:

How to enable data log on mDash

mDash can be instructed to save certain device shadow values into a persistant database. Currently, up to 100 data points can be saved. In the future, the number of datapoints will be unlimited.

In order to select which device shadow values get logged, modify the `tags.save` array, which is the array of shadow keys to log.

By default, it is set to save only `ram_free` metric. That metric gets reported by the `demo-js` default firmware.

On mDash, click on the device name. In the device shadow editor, edit the `tags` value and save. For example, to log `ram_free` and `my_value`, do:

```
...
"tags": {
  "save": [
    "ram_free",
    "my_value"
  ],
}
...
```

How to log data from a device

Device logs data via the device shadow update. Here how it is done in C/C++:

```
#include "mgos.h"
#include "mgos_shadow.h" // make sure to include "shadow" lib in mos.yml

// Somewhere in your code ...
mgos_shadow_updatef(0, "{my_value: %g}", 1.2345);
```

And in JavaScript:

```
load('api_shadow.js');

// Somewhere in your code ...
Shadow.update(0, JSON.stringify({my_value: 1.2345}));
```

See [Device shadow](#) section for the detailed device shadow explanation.

How to test data logging

You can easily simulate your device and log data from your workstation. Essentially, logging data is updating a device shadow. You can update device shadow via the REST API by sending a `POST` request to the `/devices/:id` endpoint (don't forget to change both `API_KEY` and `DEVICE_ID`):

```
curl -H 'Content-Type: application/json' \
-H 'Authorization: Bearer API_KEY' \
-d '{"shadow": {"state": {"reported": {"ram_free": 12345}}}}' \
https://dash.mongoose-os.com/api/v2/devices/DEVICE_ID
```

KIWI - a bluetooth beacon gateway

See <https://mdash.net/docs/quickstart/ble-gateway.md>

- **User guide**

Thank you for creating a pull request to contribute to Adafruit's GitHub code!

Before you open the request please review the following guidelines and tips to help it be more easily integrated:

- **Describe the scope of your change--i.e. what the change does and what parts of the code were modified.** This will help us understand any risks of integrating the code.
- **Describe any known limitations with your change.** For example if the change doesn't apply to a supported platform of the library please mention it.
- **Please run any tests or examples that can exercise your modified code.** We strive to not break users of the code and running tests/examples helps with this process.

Thank you again for contributing! We will try to test and integrate the change as soon as we can, but be aware we have many GitHub repositories to manage and can't immediately respond to every request. There is no need to bump or check in on a pull request (it will clutter the discussion of the request).

Also don't be worried if the request is closed or not integrated--sometimes the priorities of Adafruit's GitHub code (education, ease of use) might not match the priorities of the pull request. Don't fret, the open source community thrives on forks and GitHub makes it easy to keep your changes in a forked repo.

After reviewing the guidelines above you can delete this text from the pull request.

Thank you for opening an issue on an Adafruit Arduino library repository. To improve the speed of resolution please review the following guidelines and common troubleshooting steps below before creating the issue:

- **Do not use GitHub issues for troubleshooting projects and issues.** Instead use the forums at <http://forums.adafruit.com> to ask questions and troubleshoot why something isn't working as expected. In many cases the problem is a common issue that you will more quickly receive help from the forum community. GitHub issues are meant for known defects in the code. If you don't know if there is a defect in the code then start with troubleshooting on the forum first.
- **If following a tutorial or guide be sure you didn't miss a step.** Carefully check all of the steps and commands to run have been followed. Consult the forum if you're unsure or have questions about steps in a guide/tutorial.
- **For Arduino projects check these very common issues to ensure they don't apply:**
 - For uploading sketches or communicating with the board make sure you're using a **USB data cable** and **not a USB charge-only cable**. It is sometimes very hard to tell the difference between a data and charge cable! Try using the cable with other devices or swapping to another cable to confirm it is not the problem.
 - **Be sure you are supplying adequate power to the board.** Check the specs of your board and plug in an external power supply. In many cases just plugging a board into your computer is not enough to power it and other peripherals.
 - **Double check all soldering joints and connections.** Flaky connections cause many mysterious problems. See the [guide to excellent soldering](#) for examples of good solder joints.

- **Ensure you are using an official Arduino or Adafruit board.** We can't guarantee a clone board will have the same functionality and work as expected with this code and don't support them.

If you're sure this issue is a defect in the code and checked the steps above please fill in the following fields to provide enough troubleshooting information. You may delete the guideline and text above to just leave the following details:

- Arduino board: **INSERT ARDUINO BOARD NAME/TYPE HERE**
- Arduino IDE version (found in Arduino -> About Arduino menu): **INSERT ARDUINO VERSION HERE**
- List the steps to reproduce the problem below (if possible attach a sketch or copy the sketch code in too): **LIST REPRO STEPS BELOW**

Cesanta note

Imported from https://github.com/adafruit/Adafruit_SSD1306/commit/ddfec78fa15f0ff8dfc8a76524077ba6bb5fc6f3

Adafruit_SSD1306

Compatibility

MCU	Tested Works	Doesn't Work	Not Tested	Notes
Atmega328 @ 16MHz	X			
Atmega328 @ 12MHz	X			
Atmega32u4 @ 16MHz	X			
Atmega32u4 @ 8MHz	X			
ESP8266	X			change OLED_RESET to different pin if using default I2C pins D4/D5.
Atmega2560 @ 16MHz	X			
ATSAM3X8E	X			
ATSAM21D	X			
ATtiny85 @ 16MHz		X		
ATtiny85 @ 8MHz		X		
Intel Curie @ 32MHz			X	
STM32F2			X	

- ATmega328 @ 16MHz : Arduino UNO, Adafruit Pro Trinket 5V, Adafruit Metro 328, Adafruit Metro Mini

- ATmega328 @ 12MHz : Adafruit Pro Trinket 3V
- ATmega32u4 @ 16MHz : Arduino Leonardo, Arduino Micro, Arduino Yun, Teensy 2.0
- ATmega32u4 @ 8MHz : Adafruit Flora, Bluefruit Micro
- ESP8266 : Adafruit Huzzah
- ATmega2560 @ 16MHz : Arduino Mega
- ATSAM3X8E : Arduino Due
- ATSAM21D : Arduino Zero, M0 Pro
- ATTiny85 @ 16MHz : Adafruit Trinket 5V
- ATTiny85 @ 8MHz : Adafruit Gemma, Arduino Gemma, Adafruit Trinket 3V

Arduino Adafruit SSD1306 library for Mongoose OS

This library provides support for
[1-wire](#) protocol for Mongoose OS.

Certificate Authority bundle

Overview

This library provides a single file to a device's filesystem: `fs/ca.pem`.

It contains CA certificates used by the most popular services, like AWS, Google, Microsoft, etc.

RPC support for Azure

Direct Method RPC channel

Converts Azure [Direct Method](#) calls into mOS RPC calls.

Azure DM method name becomes RCP method name, payload becomes method args.

Since DM is a one way mechanism (cloud calls device), outgoing requests are rejected.

Cloud Messaging

It could be possible to support RPC over [cloud-to-device and device-to-cloud messaging](#).

Unlike DM, it could be bi-directional, with device initiating the request.

This is not implemented yet.

HTTP Server for Mongoose OS

Ethernet support

ESP32

ESP32 includes an Ethernet MAC and requires an external PHY, connected over RMII interface.

Two PHY models are currently supported:

- Microchip LAN87x0 ([LAN8710](#) supports both MII and RMII)
- [LAN8720](#) is RMII only) and [TI TLK110](#). PHY model selection is a compile-time option and is set [here](#).

There is a [detailed article](#)

on how to connect a PHY yourself. It is much easier to buy a dev board which already has one, for example,

[Olimex ESP32-EVB](#).

Once wired, `mos config-set eth.enable=true` to enable Ethernet (see below).

STM32

ESP32 includes an Ethernet MAC and requires an external PHY, connected over RMII interface.

Currently only LAN8742a PHY in RMII mode is supported.

MAC pin configuration

Since ETH MAC functions can be assigned to different pins, you need to specify pin assignments.

These are provided as C macros and can be specified in the `cdefs` section.

See example definitions for NUCLEO and Discovery boards in `mos_stm32.yml`.

Configuration

Common settings

```
"eth": {  
    "enable": true          // Enable Ethernet support  
}
```

ESP32 specific

```
"eth": {  
    "phy_addr": 0,          // RMII PHY address  
    "mdc_gpio": 23,         // GPIO to use for RMII MDC signal  
    "mdio_gpio": 18,        // GPIO to use for RMII MDIO signal  
}
```

Note: the defaults match the EVB-ESP32 board, so if you use that, you won't need to adjust anything except setting `eth.enable` to `true`.

How to contribute

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the

project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

MCP23008/MCP23017 I²C and MCP23S08/MCP23S17 SPI Driver

A Mongoose library for MCP23XXX, a popular and cheap set of GPIO extenders using either I²C (MCP320XX) or SPI (MCP32SXX)

The MCP23008 is an 8-port device, and the MCP23017 is a 16-port device, but they are otherwise identical.

NOTE: SPI driver is TODO at this point. Contact the author for details.

Implementation details

The MCP230XX is a bi-directional GPIO with input pullup capability. Each input pin can be explicitly set as INPUT or OUTPUT pin. For INPUT pins, the pin can be left floating or pulled up with a 100kOhm internal resistor.

Interrupts are handled with this driver. This allows for callbacks to be set for INPUT pin state changes.

NOTE: For simplicity reasons, on the MCP23x17 chips, `intA` and `intB` are combined, so it does not matter which of the pins are used on the chip, any port/pin state change will fire an interrupt, and only one pin needs to be used on the MCU to handle all 16 ports.

API Description

To start, `mgos_mcp23xxx_create()` is called with the correct I₂C bus and address (by default 0x20), and optionally a GPIO pin on the microcontroller that serves as an interrupt pin, to detect MCP230XX input state changes.

NOTE: When the driver starts, it polls the current state from the chip without changing any ports. The benefit of this is that the MCU can safely reboot without loss of the GPIO state in MCP230XX.

The API follows `mgos_gpio.h` closely, enabling ports to be set as input or output.

For input ports, `mgos_mcp23xxx_gpio_set_pull()` can be called with either `MGOS_GPIO_PULL_NONE` or `MGOS_GPIO_PULL_UP`, but not with `MGOS_GPIO_PULL_DOWN` which returns an error, as the chip does not support input pulldowns.

Notably, `mgos_mcp23xxx_gpio_set_int_handler()` and `mgos_mcp23xxx_gpio_set_button_handler()` work identically to the `mgos_gpio_*` variants.

Example application

```
#include "mgos.h"
#include "mgos_config.h"
#include "mgos_mcp23xxx.h"

static void button_cb(int pin, void *user_data) {
    struct mgos_mcp23xxx *d = (struct mgos_mcp23xxx *)user_data;
    LOG(LL_INFO, ("GPIO[%d] callback, value=%d", pin, mgos_mcp23xxx_gpio_read(d, pin)));
    mgos_mcp23xxx_gpio_toggle(d, pin+8);
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_mcp23xxx *d;
    int i;

    if (!(d = mgos_mcp23017_create(mgos_i2c_get_global(), mgos_sys_config_get_mcp23xxx_i2caddr(),
        mgos_sys_config_get_mcp23xxx_int_gpio())) {
        LOG(LL_ERROR, ("Could not create MCP230XX"));
        return MGOS_APP_INIT_ERROR;
    }

    for(i=0; i<4; i++) mgos_mcp23xxx_gpio_setup_input(d, i, MGOS_GPIO_PULL_UP);
    for(i=8; i<16; i++) mgos_mcp23xxx_gpio_set_mode(d, i, MGOS_GPIO_MODE_OUTPUT);

    mgos_mcp23xxx_gpio_set_button_handler(d, 0, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_NEG, 10, button_cb, d);
    mgos_mcp23xxx_gpio_set_button_handler(d, 1, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_POS, 10, button_cb, d);
    mgos_mcp23xxx_gpio_set_button_handler(d, 2, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_ANY, 10, button_cb, d);
    mgos_mcp23xxx_gpio_set_button_handler(d, 3, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_ANY, 10, button_cb, d);

    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

RPC Service - I2C

This service provides an ability to manage I2C peripherals on devices remotely. It is possible to call this service programmatically via serial, HTTP/RESTful, Websocket, MQTT or other transports (see [RPC section](#)) or via the `mos` tool.

Below is a list of exported RPC methods and arguments:

I2C.Scan

Scan the I2C bus, return a list of available device addresses. Arguments: none.

Example usage:

```
mos call I2C.Scan
[
  31    # There is only 1 device with address 31 attached to the I2C bus
]
```

I2C.Read

Read data from the I2C device. Arguments:

```
{
  "addr": 31,      // Required. Device address.
  "len": 2         // Required. Number of bytes to read.
}
```

Reply:

```
{
  // Hex-encoded data. Each byte is encoded as XX hex code, e.g. 0x00 0x1d:
  "data_hex": "001d"
}
```

Example usage:

```
mos call I2C.Read '{"addr": 31, "len": 2}'
{
  "data_hex": "001d"
}
```

I2C.Write

Write data to the I2C device. Arguments:

```
{
  "addr": 31,          // Required. Device address.
  "data_hex": "1f3c6a" // Required. Hex-encoded data to write
}
```

Example usage (showing failed write):

```
mos call I2C.Write '{"addr": 31, "data_hex": "1f3c6a"}'  
Error: remote error: I2C write failed
```

I2C.ReadRegB

Read 1-byte register value. Arguments:

```
{  
    "addr": 31, // Required. Device address.  
    "reg": 0    // Required. Register number.  
}
```

Example usage:

```
mos call I2C.ReadRegB '{"addr": 31, "reg": 0}'  
{  
    "value": 0  
}
```

I2C.WriteRegB

Write 1-byte register value. Arguments:

```
{  
    "addr": 31, // Required. Device address.  
    "reg": 0, // Required. Register number.  
    "value": 0 // Required. 1-byte value to write.  
}
```

Example usage:

```
mos call I2C.WriteRegB '{"addr": 31, "reg": 0, "value": 0}'
```

I2C.ReadRegW

Same as I2C.ReadRegB, but read 2-byte (word) register value.

I2C.WriteRegW

Same as I2C.WriteRegB, but write 2-byte (word) register value.

RPC ATCA Service support

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

Original driver

This code is derived from [Sparkfun's driver](#) which was placed in the public domain. Please consider supporting Sparkfun Electronics by picking up a few boards from them!

APDS-960 I2C Driver

A Mongoose library for Avago's APDS-9960 RGB, IR and Gesture Sensor.

Implementation details

The APDS-9960 device features advanced Gesture detection, Proximity detection, Digital Ambient Light Sense (ALS) and Color Sense (RGBC). The slim modular package, L 3.94 x W 2.36 x H 1.35 mm, incorporates an IR LED and factory calibrated LED driver for drop-in compatibility with existing footprints.

Gesture detection

Gesture detection utilizes four directional photodiodes to sense reflected IR energy (sourced by the integrated LED) to convert physical motion information (i.e. velocity, direction and distance) to a digital information. The architecture of the gesture engine features automatic activation (based on Proximity engine results), ambient light subtraction, cross-talk cancelation, dual 8-bit data converters, power saving inter-conversion delay, 32-dataset FIFO, and interrupt driven I2C communication. The gesture engine accommodates a wide range of mobile device gesturing requirements: simple UP-DOWN-RIGHT-LEFT gestures or more complex gestures can be accurately sensed. Power consumption and noise are minimized with adjustable IR LED timing.

Proximity detection

The Proximity detection feature provides distance measurement (E.g. mobile device screen to user's ear) by photodiode detection of reflected IR energy (sourced by the integrated LED). Detect/release events are interrupt driven, and occur whenever proximity result crosses upper and/ or lower threshold settings. The proximity engine features offset adjustment registers to compensate for system offset caused by unwanted IR energy reflections appearing at the sensor. The IR LED intensity is factory trimmed to eliminate the need for end-equipment calibration due to component variations. Proximity results are further improved by automatic ambient light subtraction.

Color and ALS detection

The Color and ALS detection feature provides red, green, blue and clear light intensity data. Each of the R, G, B, C channels have a UV and IR blocking filter and a dedicated data converter producing 16-bit data simultaneously. This architecture allows applications to accurately measure ambient light and sense color which enables devices to calculate color temperature and control display backlight.

API Description

There are two APIs defined in this driver. Firstly, a low level API is used to drive the configuration of the chip. Secondly, a higher level abstraction is provided for application programmers that want to merely read proximity and light data from the sensor, or install a hardware interrupt to callback user code upon certain events (such as proximity, light and gesture events described above.)

Notes

Gesture sending is incredibly hard with this sensor. Work is needed to improve the performance of the gesture code in this driver.

Proximity and Light sensing and interrupts are working fine.

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```
#include "mgos.h"
#include "mgos_config.h"
#include "mgos_apds9960.h"

static void light_interrupt(uint16_t clear, uint16_t red, uint16_t green, uint16_t blue) {
    LOG(LL_INFO, ("clear=%u red=%u green=%u blue=%u", clear, red, green, blue));
}

static void proximity_interrupt(uint8_t proximity) {
    LOG(LL_INFO, ("proximity=%u", proximity));
}

static void apds9960_timer_cb(void *user_data) {
    struct mgos_apds9960 *sensor = (struct mgos_apds9960 *)user_data;
    uint16_t clear = 0, red = 0, green = 0, blue = 0;
    uint8_t proximity = 0;

    if (!mgos_apds9960_read_light(sensor, &clear, &red, &green, &blue)) {
        LOG(LL_ERROR, ("Could not read APDS9960 light sensor"));
    }
    if (!mgos_apds9960_read_proximity(sensor, &proximity)) {
        LOG(LL_ERROR, ("Could not read APDS9960 proximity sensor"));
    }

    LOG(LL_INFO, ("clear=%u red=%u green=%u blue=%u proximity=%u", clear, red, green, blue, proximity));
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_apds9960 *sensor;

    sensor = mgos_apds9960_create(mgos_i2c_get_global(), mgos_sys_config_get_apds9960_i2caddr());
    if (!sensor) {
        LOG(LL_ERROR, ("Could not create APDS9960 sensor"));
        return false;
    }

    if (!mgos_apds9960_set_callback_light(sensor, 10, 1000, light_interrupt)) {
        LOG(LL_ERROR, ("Could not enable APDS9960 light callback"));
        return false;
    }

    if (!mgos_apds9960_set_callback_proximity(sensor, 0, 150, proximity_interrupt)) {
        LOG(LL_ERROR, ("Could not enable APDS9960 proximity callback"));
        return false;
    }

    mgos_set_timer(1000, true, apds9960_timer_cb, sensor);

    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

JS bindings for ESP32 touch pad sensor

Overview

This library provides JavaScript bindings for the ESP32 touch pad sensor.

The JS API largely mirrors the [C API](#).

Examples

Polling the sensor manually

```
load('api_esp32_touchpad.js');

// Touch sensors are numbered from 0 to 9.
// For convenience, TouchPad.GPIO map translates from GPIO number to sensor number.
let ts = TouchPad.GPIO[15];

TouchPad.init();
TouchPad.setVoltage(TouchPad.HVOLT_2V4, TouchPad.LVOLT_0V8, TouchPad.HVOLT_ATTEN_1V5);
TouchPad.config(ts, 0);
Timer.set(1000 /* 1 sec */, Timer.REPEAT, function() {
  let tv = TouchPad.read(ts);
  print('Sensor', ts, 'value', tv);
}, null);
```

Using interrupts

```
load('api_esp32_touchpad.js');

// Touch sensors are numbered from 0 to 9.
// For convenience, TouchPad.GPIO map translates from GPIO number to sensor number.
let ts = TouchPad.GPIO[15];

TouchPad.init();
TouchPad.filterStart(10);
TouchPad.setMeasTime(0x1000, 0xffff);
TouchPad.setVoltage(TouchPad.HVOLT_2V4, TouchPad.LVOLT_0V8, TouchPad.HVOLT_ATTEN_1V5);
TouchPad.config(ts, 0);
Sys.usleep(100000); // wait a bit for initial filtering.
let noTouchVal = TouchPad.readFiltered(ts);
let touchThresh = noTouchVal * 2 / 3;
print('Sensor', ts, 'noTouchVal', noTouchVal, 'touchThresh', touchThresh);
TouchPad.setThresh(ts, touchThresh);
TouchPad_isrRegister(function(st) {
  // st is a bitmap with 1 bit per sensor.
  let val = TouchPad.readFiltered(ts);
  print('Status:', st, 'Value:', val);
}, null);
TouchPad.intrEnable();
```

Mongoose OS Fingerprint Library

This library provides a simple API that describes a popular set of serial (UART) fingerprint modules made by Grow.

Application Interface

The fingerprint module consists of an ARM microcontroller and an optical or capacitive sensor unit, and it uses a Serial UART to communicate, by default at 57600 baud, with its host.

There are two modes of operation, command and data:

1. The command protocol is a serial transaction where the host writes a packet with a command, and the unit returns a packet with a response.
2. The data protocol is a repeated set of data packets, followed by an end-of-data packet.

There are two main functions that each fingerprint module exposes: enrolling fingerprints and matching fingerprints.

Enrolling fingerprints

First, two separate images are taken of the fingerprint, each stored in a memory bank. Then, they are combined into a *model* (also called a *template*), if the images are sufficiently similar. If these operations are successful, the resulting *model* can be stored into flash using a model slot.

Matching fingerprints

In this mode, an image is taken of the fingerprint, and the internal flash database is searched for a match. If a match is found, the resulting fingerprint id and quality score are returned. If no match was found (either because the fingerprint was not cleanly imaged, or if it simply does not exist in the database), an error condition is returned.

Lowlevel API primitives

First, a device is created with `mgos_fingerprint_create()`, passing in a `struct mgos_fingerprint_cfg` configuration, which can be pre-loaded with defaults by calling `mgos_fingerprint_config_set_defaults()`, after which an opaque handle is returned to the driver object. When the driver is no longer needed, `mgos_fingerprint_destroy()` can be called with a pointer to the handle, after which it will be cleaned up and memory returned.

Each API call returns a signed integer, signalling `MGOS_FINGERPRINT_OK` upon success and some other value (see `include/mgos_fingerprint.h`), negative numbers for errors, and positive numbers for successful API calls with negative results (like no finger image taken, no fingerprint found in database, etc).

Then, the following primitives are implemented:

- `mgos_fingerprint_image_get()`: This fetches an image of a fingerprint on the sensor, and stores it in RAM for further processing.
- `mgos_fingerprint_image_genchar()`: This turns the image of the fingerprint into a character array and stores it in one of two *slots*.
- `mgos_fingerprint_model_combine()`: This compares two character arrays (which represent fingerprint images), and combines them into a *model*. If the fingerprints were sufficiently similar, the operation is successful, otherwise an error is returned.
- `mgos_fingerprint_model_store()`: This stores the combined model into a position of the flash memory of the module.
- `mgos_fingerprint_database_erase()`: This erases all models from flash memory.
- `mgos_fingerprint_database_search()`: This compares an image of a fingerprint with the saved models in flash memory, returning the model number and a quality score upon success, and an error otherwise.

Some models (R502, R503, for example) may also support lighting operations:

- `mgos_fingerprint_led_on()`: This turns the LED in the sensor device on.
- `mgos_fingerprint_led_off()`: This turns the LED in the sensor device off.

- `mgos_fingerprint_led_aura()`: This uses a ring-LED around the sensor device in one of multiple colors (typically red, blue or purple) and either flashing N times, fading in or out, or swelling N times).

Library primitives

In addition to the low level primitives that the API provides, there is also a higher level implementation available. By calling `mgos_fingerprint_svc_init()`, a timer is set up at `period_ms` milliseconds intervals, at which time a fingerprint image is attempted to be created. Based on the *mode* of operation (which can be set by `mgos_fingerprint_svc_mode_set()` to either *match* or *enroll*), the fingerprint is processed accordingly.

A callback handler in `struct mgos_fingerprint_cfg` receives event callbacks as follows:

- `MGOS_FINGERPRINT_EV_INITIALIZED`: when the chip is first initialized successfully.
- `MGOS_FINGERPRINT_EV_IMAGE`: each time the sensor has successfully fetched an image.
- `MGOS_FINGERPRINT_EV_MATCH_OK`: in *match mode* each time an image matched with one of the model entries in the flash database. The matched fingerprint ID and score are packed into `*ev_data`, the top 16 bits are the `score`, the lower 16 bits are the `finger_id`.
- `MGOS_FINGERPRINT_EV_MATCH_ERROR`: in *match mode* each time an image did not match with any model entries in the flash database, or if a processing error occurred.
- `MGOS_FINGERPRINT_EV_STATE_MATCH`: when *match mode* is entered.
- `MGOS_FINGERPRINT_EV_STATE_ENROLL1`: when *enroll mode* is processing the first (of two) fingerprint images.
- `MGOS_FINGERPRINT_EV_STATE_ENROLL2`: when *enroll mode* is processing the second (of two) fingerprint images.
- `MGOS_FINGERPRINT_EV_ENROLL_OK`: when *enroll mode* successfully stored a fingerprint model in the flash database. The stored fingerprint ID is packed into `*ev_data`, the lower 16 bits are the `finger_id`.
- `MGOS_FINGERPRINT_EV_ENROLL_ERROR`: when *enroll mode* failed to process or store a fingerprint model.

Supported devices

Popular GROW devices are supported, look for Grow sensors [on Aliexpress](#).

Most of them are named **Rxxx** with a three-digit number, like **R503**,

R300 or **R307**. There is also a unit for sale at [Adafruit](#).

Example Code

For a complete demonstration of the driver, look at this [Mongoose App](#).

HD44780 library for Mongoose OS

A library used to control a HD44780 LCD display in a MongooseOS project, without an I2C module that is commonly attached to the LCD. It should use 6 GPIO pins in total, that should be mapped in config_schema like this example:

```
- ["HD44780.GPIO.D7" , 14 ]
- ["HD44780.GPIO.D6" , 12 ]
- ["HD44780.GPIO.D5" , 13 ]
- ["HD44780.GPIO.D4" , 15 ]
- ["HD44780.GPIO.E" , 4 ]
- ["HD44780.GPIO.RS" , 2 ]
- ["HD44780.PHYSICAL_NUMBER_OF_LINES" , 4 ]
```

Also, `HD44780.PHYSICAL_NUMBER_OF_LINES` is expected to be 2(default) or 4. I don't think there is models with a different number of lines.

About the other LCD pins:

- pins D0,D1,D2 and D3 are not necessary since this library uses the 4-bit interface mode.
- pin R/W should be low (GND) all the time, since this library doesn't support reading commands from display

Configuration

```
config_schema:
- ["HD44780" , "o" , {"title": "HD44780 configuration schema"}]
- ["HD44780.PHYSICAL_NUMBER_OF_LINES" , "i" , 2 , {"title": "HD44780 model available number of lines. Expected 2 or 4."}]
- ["HD44780.GPIO" , "o" , {"title": "ESP8266 GPIO pins used in HD44780's pins"}]
- ["HD44780.GPIO.D7" , "i" , 14 , {"title": "ESP8266's GPIO pin used in HD44780's D7 pin"}]
- ["HD44780.GPIO.D6" , "i" , 12 , {"title": "ESP8266's GPIO pin used in HD44780's D6 pin"}]
- ["HD44780.GPIO.D5" , "i" , 13 , {"title": "ESP8266's GPIO pin used in HD44780's D5 pin"}]
- ["HD44780.GPIO.D4" , "i" , 15 , {"title": "ESP8266's GPIO pin used in HD44780's D4 pin"}]
- ["HD44780.GPIO.E" , "i" , 4 , {"title": "ESP8266's GPIO pin used in HD44780's E pin"}]
- ["HD44780.GPIO.RS" , "i" , 2 , {"title": "ESP8266's GPIO pin used in HD44780's RS pin"}]
```

Example

This is a simple C program example using a 4-line model.

Don't forget to set `HD44780.PHYSICAL_NUMBER_OF_LINES` config equal to 4 in this case.

main.c

```

#include "mgos.h"
#include "mgos_hd44780.h"

enum mgos_app_init_result mgos_app_init(void) {

    lcd_write("First line!");

    lcd_move_cursor(19,1); // Moves to the final of the second line
    lcd_entry_mode_set( LCD_DIRECTION_DECREMENT , LCD_DISPLAY_SHIFT_OFF );
    lcd_write("Second line?"); // Writes "Second line?" backwards because entry mode decrementing cursor

    lcd_move_cursor(0,2); // Moves to the beggining of the third line
    lcd_entry_mode_set( LCD_DIRECTION_INCREMENT , LCD_DISPLAY_SHIFT_OFF );

    // Writes 1 2 3, spaced because cursor shifting
    lcd_write("1");
    lcd_shift_cursor(LCD_SHIFT_RIGHT);
    lcd_write("2");
    lcd_shift_cursor(LCD_SHIFT_RIGHT);
    lcd_write("3");

    lcd_move_cursor(9,3); // Moves to the middle of the fourth line
    lcd_write("Bye...");

    lcd_move_cursor(0,3);

    return MGOS_APP_INIT_SUCCESS;
}

```

mos.yml

```

config_schema:
  - ["HD44780.PHYSICAL_NUMBER_OF_LINES" , 4 ]

libs:
  - origin: https://github.com/rilpires/mgos_hd44780_controller
    name: hd44780

manifest_version: 2017-05-18

```

Notes

Don't forget to add:

```
name: hd44780
```

in libs section.

There is some functionalities yet to implement:

- Set the number of lines used to half of the available lines. I don't think this is useful at all but it is necessary in order to use the 5x10 font (it is basically the same font with an underline below it)
- Implement reading commands from the display. It would be necessary an additional GPIO pin.
- Implement creating custom characters.

How to contribute

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

ADS1X1X I2C Driver

A Mongoose library for various I2C speaking ADCs from Texas Instruments:

- ADS1115 - 16bit, 860 Samples/sec, 2 differential / 4 single-ended, programmable gain
- ADS1114 - 16bit, 860 Samples/sec, 1 differential / 1 single-ended, programmable gain
- ADS1113 - 16bit, 860 Samples/sec, 1 differential / 1 single-ended, no gain
- ADS1015 - 12bit, 3300 Samples/sec, 2 differential / 4 single-ended, programmable gain
- ADS1014 - 12bit, 3300 Samples/sec, 1 differential / 1 single-ended, programmable gain
- ADS1013 - 12bit, 3300 Samples/sec, 1 differential / 1 single-ended, no gain.

The most common are the **ADS1115** and **ADS1015** chips.

Implementation details

The driver takes care of exposing the correct functionality based on which type is created. Differential measurements can be taken on all devices, but only ADS1x15 has multiple options.

API Description

First, create a device using `mgos_ads1x1x_create()` by specifying the type of chip you're using. Take some measurements using `mgos_ads1x1x_read()`, and clean up the driver by using `mgos_ads1x1x_destroy()`.

`mgos_ads1x1x_set_fsr()` is used to set the full scale range (FSR) of the ADC. Each chip supports ranges from 6.144 Volts down to 0.256 Volts. You can read the current FSR with `mgos_ads1x1x_get_fsr()`.

`mgos_ads1x1x_set_dr()` is used to set the data rate of continuous measurements. The support differs between ADS101X (the 12-bit version, which is faster), and ADS111X (the 16-bit version, which is slower). You can read the current DR with `mgos_ads1x1x_get_dr()`.

`mgos_ads1x1x_read()` starts a singleshot measurement on the given channel (which takes 1ms for ADS101X and 8ms for ADS111X), and returns a 16 bit signed value. The datasheet mentions that with input voltages around GND, a negative value might be returned (ie -2 rather than 0).

`mgos_ads1x1x_read_diff()` starts a singleshot measurement of the differential voltage between two channels, typically Chan0 and Chan1. Several channel pairs are allowed, see the include file for details. Note, that this function is only available on ADS1X15 chips.

Example application

```

#include "mgos.h"
#include "mgos_config.h"
#include "mgos_ads1x1x.h"

void timer_cb(void *data) {
    struct mgos_ads1x1x *d = (struct mgos_ads1x1x *)data;
    int16_t res[4];

    if (!d) return;

    for(int i=0; i<4; i++) {
        if (!mgos_ads1x1x_read(s_adc, i, &res[i])) {
            LOG(LL_ERROR, ("Could not read device"));
            return;
        }
    }
    LOG(LL_INFO, ("chan=%{#6d, #6d, #6d, #6d}", res[0], res[1], res[2], res[3]));
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_ads1x1x *d = NULL;

    if (!(d = mgos_ads1x1x_create(mgos_i2c_get_global(), 0x48, ADC_AMPLIFIER))) {
        LOG(LL_ERROR, ("Could not create ADS1115"));
        return MGOS_APP_INIT_ERROR;
    }

    mgos_set_timer(100, true, timer_cb, d);

    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Use mDash API to set a device label and create the UI

Example 1

```

#include "mgos.h"

#define USE_WIFI 0

#if USE_WIFI == 1
#include "mgos_wifi.h"
#endif

#include "mgos_mdash_api.h"

static void create_mdash_ui() {
    struct mgos_mdash_widgets *widgets = mgos_mdash_widgets_create(0);
    mgos_mdash_widgets_add_widget(
        widgets, mgos_mdash_widget_toggle_create("Switch LED on/off",
                                                "state.reported.on"));
    mgos_mdash_widgets_add_widget(
        widgets,
        mgos_mdash_widget_value_create("Uptime (s):", "state.reported.uptime"));
    mgos_mdash_widgets_add_widget(
        widgets, mgos_mdash_widget_input_create(
            "Led pin", "state.reported.led_pin", "fa-save"));
    mgos_mdash_widgets_add_widget(
        widgets, mgos_mdash_widget_button_create("Reboot", "Sys.Reboot", NULL,
                                                "fa-power-off"));

    mgos_mdash_create_ui(widgets);
    mgos_mdash_widgets_free(widgets);
}

#if USE_WIFI == 1
static void wifi_cb(int ev, void *evd, void *arg) {
    switch (ev) {
        case MGOS_WIFI_EV_STA_IP_ACQUIRED: {
            LOG(LL_INFO, ("%s", "MGOS_WIFI_EV_STA_IP_ACQUIRED"));

            mgos_mdash_set_label("my_label");
            create_mdash_ui();

            break;
        }
    }
    (void) evd;
    (void) arg;
}
#else
static void net_cb(int ev, void *evd, void *arg) {
    switch (ev) {
        case MGOS_NET_EV_IP_ACQUIRED: {
            LOG(LL_INFO, ("%s", "MGOS_NET_EV_IP_ACQUIRED"));

            mgos_mdash_set_label("my_label");
            create_mdash_ui();
            break;
        }
    }
    (void) evd;
    (void) arg;
}
#endif

enum mgos_app_init_result mgos_app_init(void) {
#if USE_WIFI == 1
    mgos_event_add_group_handler(MGOS_EVENT_GRP_WIFI, wifi_cb, NULL);
#else
    mgos_event_add_group_handler(MGOS_EVENT_GRP_NET, net_cb, NULL);
#endif

    return MGOS_APP_INIT_SUCCESS;
}

```

Example 2

- Define the widgets in the `mos.yml` of your application, e.g.

```
config_schema:  
# mDash widgets  
- ["mdash.toggle.title", "Switch LED on/off"]  
- ["mdash.toggle.key", "state.reported.on"]  
  
- ["mdash.value.title", "Uptime (s):"]  
- ["mdash.value.key", "state.reported.uptime"]  
  
- ["mdash.value1.enable", true]    # enable this widget  
- ["mdash.value1.title", "Min free RAM:"]  
- ["mdash.value1.key", "state.reported.ram_min_free"]  
  
- ["mdash.value2.enable", true]    # enable this widget  
- ["mdash.value2.title", "Free RAM:"]  
- ["mdash.value2.key", "state.reported.ram_free"]  
  
- ["mdash.value3.enable", true]    # enable this widget  
- ["mdash.value3.title", "Temp:"]  
- ["mdash.value3.key", "state.reported.temp"]  
  
- ["mdash.button.title", "Reboot"]  
- ["mdash.button.method", "Sys.Reboot"]  
- ["mdash.button.params", ""]  
- ["mdash.button.icon", "fa-power-off"]
```

- The code

```
#include "mgos.h"  
  
#include "mgos_mdash_api.h"  
  
static void create_mdash_ui() {  
    struct mgos_mdash_widgets *widgets;  
    if (mgos_mdash_create_widgets_from_config(&widgets)) {  
        mgos_mdash_create_ui(widgets);  
    }  
    mgos_mdash_widgets_free(widgets);  
}  
  
  
static void cloud_cb(int ev, void *evd, void *arg) {  
    switch (ev) {  
        case MGOS_EVENT_CLOUD_CONNECTED: {  
            LOG(LL_INFO, ("%s - Cloud connected", __FUNCTION__));  
            mgos_mdash_set_label("my_label");  
            create_mdash_ui();  
            break;  
        }  
    }  
    (void) evd;  
    (void) arg;  
}  
  
enum mgos_app_init_result mgos_app_init(void) {  
    mgos_event_add_handler(MGOS_EVENT_CLOUD_CONNECTED, cloud_cb, NULL);  
  
    return MGOS_APP_INIT_SUCCESS;  
}
```

- Drawback - the order of creating the widgets is fixed: value, input, toggle and button.
- Advantage - a widget can be disabled at runtime. E.g.

```
mos --port <serial,ws,mqtt port> config-set mdash.value3.enable=false
```

Usage

- Create a device in mDash
- Create a API key
- Build and flash the application
- mos wifi SSID PASSWORD
- Provision the device in mDash
- mos --port <port> config-set mdash.device_id="" mdash.api_key=""

Refs

[Device UI for remote control](#)

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing

tests if you're unsure how to go about it.

4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
 5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
 6. Finally, push the commits to your fork and submit a [pull request](#).
- Software License Agreement (BSD License)

Copyright (c) 2012 Adafruit Industries. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Attribution

These are files taken from Adafruit's GFX library:

<https://github.com/adafruit/Adafruit-GFX-Library>

Modifications

- `gfont.h`: added a few attributes to `struct GFXfont`
- `fonts/*.h`: added the attributes from `struct GFXfont`
- `fonts/*.h`: wrapped the data in `#ifndef X; #define X; #endif` stanzas
- `src/mgos ili9341_primitives.c`: Rewrote from C++ class to standard C implementation. Also, ensured each function takes (x,y) arguments that are unsigned ints, as this implementation sets a window to draw in.

Attribution

This is a copy of uPNG taken from <https://github.com/elanthis/upng>

Modifications

None.

ILI9341 TFT Driver

Introduction

This library is a native implementation of a graphics chip for small TFT screens, the ILI9341. The chip is found in many popular displays, including 320x240 and 480x320 pixel versions, and is capable of driving 16-bit (RGB-565) as well as 18-bit (RGB-666) modes. Although the chip is capable of using 18-bit colors (262,114), that requires three bytes per pixel to be written. Therefore, this driver initializes it in 16-bit colors (65,536) allowing for two bytes per pixel.

Fundamentals

The chip allows access to its framebuffer memory by setting a window by means of (x0,y0)-(x1,y1) coordinates, and then writing pixel data. To implement higher level shapes and fonts, we implement three basic functions on the hardware:

```
void mgos_ilis341_drawPixel(uint16_t x0, uint16_t y0);
```

This function sets the window to be exactly 1 pixel and writes the current foreground color to it.

```
void mgos_ilis341.drawLine(uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1);
```

This function can draw a vertical line by setting the window to (x0,y0)-(x0+1,y1), and writing the current foreground color to it. It can draw a horizontal line similarly by setting the window to (x0,y0)-(x1,y0+1). Any other lines can be decomposed into segments of horizontal and vertical lines and single pixels.

```
void mgos_ilis341.fillRect(uint16_t x0, uint16_t y0, uint16_t w, uint16_t h);
```

This sets the window to (x0,y0)-(x0+w-1,y0+h-1) and writes the current foreground color to it.

Primitives

Orientations

```
void mgos_ilis341.set_orientation(uint8_t madctl, uint16_t rows, uint16_t cols);
```

Depending on how the hardware manufacturer connected the LCD panel to the ILI9341 chip, several registers are provided to determine the true orientation of the screen. From the datasheet, there are 5 bits which determine that orientation, in the MADCTL register, as follows:

- bit2 ILI9341_MADCTL_MH - sets the *Horizontal Refresh*, 0=Left-Right and 1=Right-Left
- bit4 ILI9341_MADCTL_ML - sets the *Vertical Refresh*, 0=Top-Bottom and 1=Bottom-Top

- bit5 `ILI9341_MADCTL_MV` - sets the *Row/Column Swap*, 0=Normal and 1=Swapped
- bit6 `ILI9341_MADCTL_MX` - sets the *Column Order*, 0=Left-Right and 1=Right-Left
- bit7 `ILI9341_MADCTL_MY` - sets the *Row Order*, 0=Top-Bottom and 1=Bottom-Top

This bits are defined as `ILI9341_MADCTL_*` in `mgos ili9341.h` header file.

By means of example, here's a definition for Adafruit panels:

```
#define ADAFRUIT_PORTRAIT      (ILI9341_MADCTL_MX)
#define ADAFRUIT_LANDSCAPE     (ILI9341_MADCTL_MX|ILI9341_MADCTL_MY|ILI9341_MADCTL_MV)
#define ADAFRUIT_PORTRAIT_FLIP (ILI9341_MADCTL_MY)
#define ADAFRUIT_LANDSCAPE_FLIP (ILI9341_MADCTL_MV)
```

And here's a definition for the panel used in M5Stack:

```
#define M5STACK_PORTRAIT      (ILI9341_MADCTL_MV|ILI9341_MADCTL_MY)
#define M5STACK_LANDSCAPE     (0x00)
#define M5STACK_PORTRAIT_FLIP (ILI9341_MADCTL_MV|ILI9341_MADCTL_MX)
#define M5STACK_LANDSCAPE_FLIP (ILI9341_MADCTL_MY|ILI9341_MADCTL_ML|ILI9341_MADCTL_MX)
```

The API call `mgos ili9341_set_orientation()` gives full control over these settings, as well as setting the resulting width and height in pixels.

Window and Clipping

The driver works by setting a bounding box around the area to be drawn in:

```
void mgos ili9341_set_window(uint16_t x0, uint16_t y0,
                           uint16_t x1, uint16_t y1);
```

Any operation will be homed within the window, and clipped if it runs outside of the window. For example, setting the window to (20,30)-(99,99) will create a bounding box of 80 pixels wide and 70 pixels high. Subsequently calling `mgos ili9341_drawPixel(0,0)` will draw a pixel at the top left of the window, which in this case is physical pixel (20,30). Drawing a pixel at coordinates (80,70) will draw a pixel at the bottom right corner of the window, whereas drawing one at (81,71) will not show anything.

Geometric Shapes

A set of primitives are provided to allow drawing of geometric shapes such as circles, triangles and boxes:

```
void mgos ili9341_drawRect(uint16_t x0, uint16_t y0, uint16_t w, uint16_t h);
void mgos ili9341_drawRoundRect(uint16_t x0, uint16_t y0, uint16_t w,
                                uint16_t h, uint16_t r);
void mgos ili9341_fillRoundRect(uint16_t x0, uint16_t y0, uint16_t w,
                                 uint16_t h, uint16_t r);

void mgos ili9341_drawCircle(uint16_t x0, uint16_t y0, uint16_t r);
void mgos ili9341_fillCircle(uint16_t x0, uint16_t y0, uint16_t r);

void mgos ili9341_drawTriangle(uint16_t x0, uint16_t y0, uint16_t x1,
                               uint16_t y1, uint16_t x2, uint16_t y2);
void mgos ili9341_fillTriangle(uint16_t x0, uint16_t y0, uint16_t x1,
                               uint16_t y1, uint16_t x2, uint16_t y2);
```

Fonts

Fonts can be embedded in the program by including the font files in `fonts/*.h`.

The width and height of a given string can be calculated, and a string can be

printed (using the set foreground and background colors). As with geometric shapes, the print function is relative to the window, (0,0) being top-left, and will be clipped at the window borders.

Images

Since the ILI9341 chip expects pixels in 16-bit network byte order, we can transmit a line of pixels from an image. Because the build target of the driver is all Mongoose available platforms (notably ESP8266, ESP32 and CC32xx), memory is an issue, and decompressing PNG, GIF, or JPG files can be CPU as well as memory intensive, a simplistic alternative is provided: DIF

This function can be used for displaying DIF images:

```
void mgos_ilis341_drawDIF(uint16_t x0, uint16_t y0, char *fn);
```

DIF file format

DIF stands for **dumb image format**, and consists of a 16 byte header followed by raw pixel data, 16 bits per pixel, in network byte order. This alleviates CPU and memory pressure on the micro controller, at the expense of file size. The header is as follows:

- The first four bytes (offset 0..3) are the string DIF\001
- The second four bytes (offset 4..7) are the width (an `uint32_t` in network byte order).
- The third four bytes (offset 8..11) are the height (an `uint32_t` in network byte order).
- The byte at offset 12 denotes the raw image format that follows:
 - Value of 0 means: 16 bit RGB-565 in network byte order.
- The last three bytes (offset 13..15) are reserved for future use.

The file size will therefore be `w * h + 16` bytes.

Converting PNG to DIF

To convert PNG images to DIF images, there is a `png2dif` utility available in this repo, under `contrib/png2dif`

Clone this repo, and while inside the `contrib/png2dif` directory, run `make` to generate `png2dif` executable. You can then run `png2dif -i input.png -o output.dif` to convert a non-transparent png file, to a dif file.

The generated dif file should then be placed inside the fs directory, and can be displayed using `mgos_ilis341_drawDIF`. As an example, if the filename is `output.dif`, to display at x cord of 0 and y cord of 0:

```
mgos_ilis341_drawDIF( 0, 0, "/output.dif" );
```

Example usage of DIF images, and this library, can be found in the [Huzzah Featherwing Example App](#)

Example Application

mos.yml

The driver uses the Mongoose native SPI driver. It is configured by setting up the MOSI, MISO, SCLK pins and assigning one of the three available CS positions, in this example (which was taken from the Huzzah32 ESP32 microcontroller), we are going to use CS0:

```
config_schema:  
- ["spi.enable", true]  
- ["spi.cs0_gpio", 15]    # The ILI9341 CS pin  
- ["spi.cs1_gpio", -1]  
- ["spi.cs2_gpio", -1]  
- ["spi.mosi_gpio", 18]  
- ["spi.miso_gpio", 19]  
- ["spi.sclk_gpio", 5]  
- ["ili9341.cs_index", 0] # Use spi.cs0_gpio  
- ["ili9341.dc_pin", 33]
```

Application

```
#include "mgos.h"  
#include "mgos ili9341.h"  
#include "fonts/FreeMonoBold9pt7b.h"  
  
enum mgos_app_init_result mgos_app_init(void) {  
    mgos_ili9341_set_fgcolor(0xff, 0, 0);           // Red  
    mgos_ili9341_set_window(20, 30, 119, 59);        // 100x30 pixels  
    mgos_ili9341_drawRoundRect(0, 0, 100, 30, 8);   // Draw a rounded rectangle  
    mgos_ili9341_set_fgcolor(0, 0xff, 0);            // Green  
    mgos_ili9341_set_font(&FreeMonoBold9pt7b);      // Set font  
    mgos_ili9341_print(5, 5, "Hello World");  
  
    return MGOS_APP_INIT_SUCCESS;  
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

This library implements a [analog-to-digital conversion](#) for Mongoose OS.

Blynk integration for Mongoose OS

This Mongoose OS library allows your device remote control via the Blynk platform. Device side logic could be implemented in either C/C++ or JavaScript.

Blynk is a platform with iOS and Android apps to control Arduino, Raspberry Pi and the likes over the Internet.

See example video at:



How to use this library

In your Mongoose OS app, edit `mos.yml` file and add a reference to this library. See an [example blynk app](#) that does that.

Device configuration

This library adds `blynk` configuration section to the device:

```
mos config-get blynk
{
  "auth": "YOUR_BLYNK_AUTH_TOKEN",
  "enable": true,
  "server": "blynk-cloud.com:8442"
}
```

In order for your device to authenticate with Blynk cloud, either use Web UI to change the `blynk.auth` value, or in a terminal:

```
mos config-set blynk.auth=YOUR_BLYNK_AUTH_TOKEN
```

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate](#)

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

INA226 I2C Driver

A Mongoose library for Texas Instruments' popular I2C current sensor.

Implementation details

The [INA226](#) is a simple I2C device that measures a voltage drop over a shunt resistor, as well as the voltage on the bus. It is able to measure voltage up to 36V (in 1.25mV increments) on the bus and the voltage drop over the shunt resistor in 2.5uV increments with a range of 81.92mV, and is very precise.

Using a 0.1Ohm shunt resistor, the maximum current that can be measured is 0.8A.

API Description

Create an [INA226](#) object using `mgos_ina226_create()`, set the shunt resistor value using `mgos_ina226_set_shunt_resistance()`, then call `mgos_ina226_get_bus_voltage()` to get the bus voltage, call `mgos_ina226_get_shunt_voltage()` to get the voltage drop over the shunt resistor, and `mgos_ina226_get_current()` to get the current flowing through the circuit.

Example application

```
#include "mgos.h"
#include "mgos_config.h"
#include "mgos_ina226.h"

static void ina226_timer_cb(void *user_data) {
    struct mgos_ina226 *sensor = (struct mgos_ina226 *)user_data;
    float bus, shunt, current, res;

    if (!sensor) return;

    mgos_ina226_get_bus_voltage(sensor, &bus);
    mgos_ina226_get_shunt_resistance(sensor, &res);
    mgos_ina226_get_shunt_voltage(sensor, &shunt);
    mgos_ina226_get_current(sensor, &current);
    LOG(LL_INFO, ("Vbus=% .3f V Vshunt=% .0f uV Rshunt=% .3f Ohm Ishunt=% .1f mA",
        bus, shunt*1e6, res, current*1e3));
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_ina226 *sensor;

    sensor = mgos_ina226_create(mgos_i2c_get_global(), mgos_sys_config_get_ina226_i2caddr());
    if (!sensor) {
        LOG(LL_ERROR, ("Could not create INA226 sensor"));
        return false;
    }

    mgos_set_timer(1000, true, ina226_timer_cb, sensor);

    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Crontab

Crontab wraps [cron core](#) and maintains a persisted set of cron jobs. Crontab file is simply a JSON file (actually managed by [jstore](#)) which looks like this:

```
{"items": [
    ["1", {
        "at": "0 0 7 * * MON-FRI",
        "enable": true,
        "action": "foo",
        "payload": {"a": 1, "b": 2}
    }],
    ["2", {
        "at": "0 */2 1-4 * * *",
        "enable": true,
        "action": "bar"
    }]
]}
```

For the cron expression syntax, see [cron](#)
[core](#) docs.

Arduino compatibility for Mongoose OS

This library provides a common Arduino compatibility layer, so that one could pick an existing Arduino program, throw it into the Mongoose OS application sources, and ideally, it "just works".

Currently, the following public headers are provided:

- `Arduino.h`
- `Print.h`
- `WString.h`
- `stdlib_nonoiso.h`

There are more specific Arduino-compatibility libraries available: for
[onewire](#),
[SPI](#), etc.

A Bosch BM222 accelerometer library

Overview

This is a library for the Bosch BM222 accelerometer.

See https://www.bosch-sensortec.com/bst/products/all_products/bma222e for
more information about the hardware. VFS subsystem multiplexes calls to libc file API methods such as open,
read, write and close between (potentially) several filesystems attached
at different mount points.

A filesystem is backed by a device which supports block reads and writes.

Generic MQTT client

This library provides [MQTT protocol](#) client
API that allows devices to talk to MQTT servers.

Mongoose OS implements MQTT 3.1.1 client functionality, and works with
all popular MQTT server implementations, like AWS IoT, Google IoT Core,
Microsoft Azure, IBM Watson, HiveMQ, Mosquitto, etc.

In order to talk to an MQTT server, configure MQTT server settings -
see Configuration section below. Once configured, Mongoose OS keeps that
connection alive by reconnecting and re-subscribing to all topics
after disconnections - you do not need to implement the reconnection logic.

If you want to use TLS, set `mqtt.ssl_ca_cert=ca.pem`. Make sure that `ca.pem`
file has required CA certificates. If you want to use mutual TLS, set
`mqtt.ssl_cert=CLIENT_CERT.pem` and `mqtt.ssl_key=PRIVATE_KEY.pem`.

See example video (don't forget to set `mqtt.enable=true` before you try it):

Configuration

The MQTT library adds `mqtt` section to the device configuration:

```
{  
    "clean_session": true,           // Clean session info stored on server  
    "client_id": "",                // If not set, device.id is used  
    "enable": false,                 // Enable MQTT functionality  
    "keep_alive": 60,               // How often to send PING messages in seconds  
    "pass": "",                     // User password  
    "reconnect_timeout_min": 2,      // Minimum reconnection timeout in seconds  
    "reconnect_timeout_max": 60,     // Maximum reconnection timeout in seconds  
    "server": "iot.eclipse.org",    // Server to connect to. if `:PORT` is not specified,  
                                // 1883 or 8883 is used depending on whether SSL is enabled.  
    "ssl_ca_cert": "",              // Set this to file name with CA certs to enable TLS  
    "ssl_cert": "",                // Client certificate for mutual TLS  
    "ssl_cipher_suites": "",        // TLS cipher suites  
    "ssl_key": "",                  // Private key for the client certificate  
    "ssl_psk_identity": "",         // If set, a preshared key auth is used  
    "ssl_psk_key": "",              // Preshared key  
    "user": "",                     // MQTT user name, if MQTT auth is used  
    "will_message": "",             // MQTT last will message  
    "will_topic": ""               // MQTT last will topic  
}
```

Reconnect behavior and backup server

It is possible to have a "backup" server that device will connect to if it fails to connect to the primary server.

Backup server is configured under the `mqtt1` section which contains exactly the same parameters as `mqtt` described above.

Device will first try to connect to the main server configured under `mqtt`.

It will keep connecting to it, increasing the reconnection interval from `reconnect_timeout_min` to `reconnect_timeout_max`.

Reconnection interval is doubled after each attempt so for values above there will be connection attempts after 2, 4, 8, 16, 32 and 60 seconds.

After reaching the maximum reconnect interval and if `mqtt1.enable` is set, it will switch to the `mqtt1` configuration and reset the reconnect interval, so it will try to connect to `mqtt1` the same way.

If that works, it will stay connected to `mqtt1`. If connection drops, it will try to reconnect to `mqtt1` in the same way. If connection to backup server fails, it will go back to the main server and so on.

Arduino OneWire library for Mongoose OS

This library provides an Arduino compatibility layer for onewire by providing

an `OneWire.h` public header, so that one could pick an existing Arduino program which uses `onewire`, throw it into the Mongoose OS application sources, and ideally, it "just works".

Additionally, a mgos-specific API is available, see `include/mgos_arduino_onewire.h` and `mjs_fs/api_arduino_onewire.js`.

DS18B20 Library for the ESP32# Azure IoT Hub support library for Mongoose OS

This library provides [Azure IoT Hub](#) support for Mongoose OS.

Currently only plain MQTT is supported.

See Azure IoT + Mongoose OS tutorial at <https://mongoose-os.com/docs/quickstart/cloud/azure.md>

Authentication

Authentication by both SAS token and X.509 certificate is supported. See the [Authentication section](#) of the documentation for explanation.

`mos azure-iot-setup`

The easiest way to setup Azure cloud connection is by using `mos azure-iot-setup`. Makes sure you have the `az` CLI tool installed, create an IoT Hub, then run:

```
$ mos azure-iot-setup --azure-hub-name MY-HUB-NAME --azure-device-id NEW-DEVICE-ID
```

SAS Token

To use symmetric key authentication, obtain the connection string from the web interface or by using the `az` CLI utility:

```
$ az iot hub device-identity show-connection-string --hub-name my-hub --device-id test1
{
  "cs": "HostName=my-hub.azure-devices.net;DeviceId=test1;SharedAccessKey=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA="
}
```

Enable the Azure client and set the `azure.cs` config setting:

```
$ mos config-set azure.enable=true "azure.cs=HostName=my-hub.azure-devices.net;DeviceId=test1;SharedAccessKey=AAAAAAAAAAAAAAAAAAAAAAA="
```

X.509 Certificate

To use authentication by an X.509 certificate, upload the certificate and private key files in PEM format on the device and configure `azure.host_name`, `azure.device_id`, `azure.cert` and `azure.key`:

```
$ mos put test4.crt.pem
$ mos put test4.key.pem
$ mos config-set azure.enable=true azure.host_name=my-hub.azure-devices.net azure.device_id=test4 \
  azure.cert=test4.crt.pem azure.key=test4.key.pem
```

Note: It is possible to store private key in a cryptochip, such as [ATECC508A](#) (for example, as described [here](#) for Google IoT Core). Just specify `azure.key=ATCA:0` to use private key in slot 0 of the chip. [mos `azure-iot-setup`] supports ATECC508 key storage - just add `--use-atca` to the setup command above.

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

CCS811 I2C Driver

A Mongoose library for AMS CCS811 integrated circuit.

Sensor details

The CCS811 is an ultra-low power digital gas sensor solution which integrates a metal oxide (MOX) gas sensor to detect a wide range of Volatile Organic Compounds (VOCs) for indoor air quality monitoring with a microcontroller unit (MCU), which includes an Analog-to-Digital converter (ADC), and an I2C interface.

CCS811 is based on ams unique micro-hotplate technology which enables a highly reliable solution for gas sensors, very fast cycle times and a significant reduction in average power consumption.

The integrated MCU manages the sensor driver modes and measurements. The I2C digital interface significantly simplifies the hardware and software design, enabling a faster time to market.

CCS811 supports intelligent algorithms to process raw sensor measurements to output a TVOC value or equivalent CO₂ (eCO₂) levels, where the main cause of VOCs is from humans.

CCS811 supports multiple measurement modes that have been optimized for low-power consumption during an active sensor measurement and idle mode extending battery life in portable applications.

See [datasheet](#)

for implementation details.

A great place to buy a ready made and tested unit is at [Adafruit](#).

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```

#include "mgos.h"
#include "mgos_i2c.h"
#include "mgos_ccs811.h"

static struct mgos_ccs811 *s_ccs811;

static void timer_cb(void *user_data) {
    float eco2, tvoc;

    eco2=mgos_ccs811_get_eco2(s_ccs811);
    tvoc=mgos_ccs811_get_tvoc(s_ccs811);

    LOG(LL_INFO, ("ccs811 eCO2=%f ppm TVOC=%f ppb", eco2, tvoc));

    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_i2c *i2c;

    i2c=mgos_i2c_get_global();
    if (!i2c) {
        LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
    } else {
        s_ccs811=mgos_ccs811_create(i2c, 0x5a); // Default I2C address
        if (s_ccs811) {
            mgos_set_timer(5000, true, timer_cb, NULL);
        } else {
            LOG(LL_ERROR, ("Could not initialize sensor"));
        }
    }
    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Miniz compression library

Overview

Provides support for reading and writing ZIP archives.

This is the version 2.0.7 of the library, obtained from [here](#).

License

MIT 3-clause:

All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

OTA firmware update via the device shadow

Overview

This library provides a way to update firmware Over-The-Air via the device shadow mechanism.

In the text below, AWS IoT Shadow will be used.

It works by observing the `ota.url` entry in the device shadow. That entry must be an URL with the valid Mongoose OS firmware, which is a .zip file. When that entry is changed, this library downloads a firmware from that URL and kicks off an OTA update. The sequence of actions is as follows:

- Receive shadow delta for `ota.url` - a new `ota.url` value
- Compare `ota.url` with file on flash. If it is the same as new value, stop
- If it is different, save new `ota.url` locally on flash
- Report new `ota_url`
- Trigger an OTA by downloading the .zip
- During the process, report numeric `ota.code` and string `ota.message` entries which indicates the status of the OTA in nearly real time
- On any failure, stop with the failure `ota.message`
- On success, reboot to the new firmware
- After reboot, commit the new firmware after the successful AWS IoT handshake
- If AWS IoT handshake does not happen during 5 minutes (the default commit timeout), rollback to the old firmware
- OTA failure keeps the `ota.url` delta uncleared

How to trigger an OTA firmware update

Build and flash the <https://github.com/mongoose-os-apps/demo-js> app, start the console, and provision to AWS IoT. Open the shadow GUI for the device.

Upload built firmware to the

Mongoose OS dashboard

and copy the firmware URL.

Change the desired shadow state by adding a section `ota` with `url` key:

```
{  
  "desired": {  
    "ota": {  
      "url": "https://mongoose-os.com/dashboard/ota/PATH/TO/APP.zip"  
    },  
    ...  
  },  
  ...  
}
```

You can edit device shadow from the AWS IoT dashboard:

When it successfully finishes, it reboots and commits the firmware after the AWS IoT handshake:

PPPoS / cellular modem support

This library provides IP over serial port. Encapsulation is PPP.

Settings

```
"pppos": {  
  "enable": false,          # Enable PPPoS  
  "uart_no": 1,            # Which UART to use.  
  "baud_rate": 115200,     # Baud rate, data mode is 8-N-1.  
  "fc_enable": false,       # Enable hardware CTS/RTS flow control  
  "apn": "",               # APN name  
  "user": "",               # User name  
  "pass": "",               # Password  
  "connect_cmd": "ATDT*99***1#", # AT command to send to initiate PPP data mode  
  "echo_interval": 10,       # LCP Echo interval, seconds  
  "echo_fails": 3,           # Number of failed echos before connection is declared dead are retried  
  "hexdump_enable": false    # Dump all the data sent over UART to stderr  
}
```

Default UART pin assignments are used and they can be found [here](#).

Example configuration

Access Point Name, PPP username and password depend on the operator. They are usually public and can be found [here](#).

Here's an example for Vodafone Ireland:

```
"pppos": {  
  "enable": true,  
  "uart_no": 1,  
  "apn": "live.vodafone.ie",  
  "user": "dublin",  
  "pass": "dublin",  
}
```

RPC Service - WiFi

Wifi.Scan

Scan wifi networks.

Arguments: none.

Example usage:

```
mos call Wifi.Scan
[
  {
    "ssid": "my_essid",
    "bssid": "12:34:56:78:90:ab",
    "auth": 0,
    "channel": 1,
    "rssi": -25
  },
  ...
]
```

Authentication ENUM mapping

From mgos_wifi.h

```
enum mgos_wifi_auth_mode {
  MGOS_WIFI_AUTH_MODE_OPEN = 0,
  MGOS_WIFI_AUTH_MODE_WEP = 1,
  MGOS_WIFI_AUTH_MODE_WPA_PSK = 2,
  MGOS_WIFI_AUTH_MODE_WPA2_PSK = 3,
  MGOS_WIFI_AUTH_MODE_WPA_WPA2_PSK = 4,
  MGOS_WIFI_AUTH_MODE_WPA2_ENTERPRISE = 5,
};
```

README for git hooks script

git has a way to run scripts, which are invoked by specific git commands.

The git hooks are located in <mbed TLS root>/.git/hooks, and as such are not under version control
for more information, see the [git documentation](#).

The mbed TLS git hooks are located in <mbed TLS root>/tests/git-scripts directory, and one must create a soft link from <mbed TLS root>/.git/hooks to <mbed TLS root>/tesst/git-scripts, in order to make the hook scripts successfully work.

Example:

Execute the following command to create a link on linux from the mbed TLS .git/hooks directory:

```
In -s ../../tests/git-scripts/pre-push.sh pre-push
```

Note: Currently the mbed TLS git hooks work only on a GNU platform. If using a non-GNU platform, don't enable these hooks!

These scripts can also be used independently.

Note: This is just a template, so feel free to use/remove the unnecessary things

Description

- Type: Bug | Enhancement\Feature Request | Question

- Priority: Blocker | Major | Minor
-

Bug

OS

Mbed OS|linux|windows|

mbed TLS build:

Version: x.x.x or git commit id

OS version: x.x.x

Configuration: please attach config.h file where possible

Compiler and options (if you used a pre-built binary, please indicate how you obtained it):

Additional environment information:

Peer device TLS stack and version

OpenSSL|GnuTLS|Chrome|NSS(Firefox)|SecureChannel (IIS/Internet Explorer/Edge)|Other

Version:

Expected behavior

Actual behavior

Steps to reproduce

Enhancement\Feature Request

Justification - why does the library need this feature?

Suggested enhancement

Question

Please first check for answers in the [Mbed TLS knowledge Base](#), and preferably file an issue in the [Mbed TLS support forum](#)

Notes:

- Pull requests cannot be accepted until:
- The submitter has [accepted the online agreement here](#) with a click through
or for companies or those that do not wish to create an mbed account, a slightly different agreement can be found [here](#)
- The PR follows the [mbed TLS coding standards](#)
- This is just a template, so feel free to use/remove the unnecessary things

Description

A few sentences describing the overall goals of the pull request's commits.

Status

READY/IN DEVELOPMENT/HOLD

Requires Backporting

When there is a bug fix, it should be backported to all maintained and supported branches.

Changes do not have to be backported if:

- This PR is a new feature\enhancement
- This PR contains changes in the API. If this is true, and there is a need for the fix to be backported, the fix should be handled differently in the legacy branch

Yes | NO

Which branch?

Migrations

If there is any API change, what's the incentive and logic for it.

YES | NO

Additional comments

Any additional information that could be of interest

Todos

- Tests
- Documentation
- Changelog updated
- Backported

Steps to test or reproduce

Outline the steps to test or reproduce the PR here. Contributing

We gratefully accept bug reports and contributions from the community. There are some requirements we need to fulfill in order to be able to integrate contributions:

- As with any open source project, contributions will be reviewed by the project team and community and may need some modifications to be accepted.
- The contribution should not break API or ABI, unless there is a real justification for that. If there is an API change, the contribution, if accepted, will be merged only when there will be a major release.

Contributor License Agreement (CLA)

- All contributions, whether large or small, require a Contributor's License Agreement (CLA) to be accepted. This is because source code can possibly fall under copyright law and we need your consent to share in the ownership of the copyright.
- To accept the Contributor's License Agreement (CLA), individual contributors can do this by creating an Mbed account and [accepting the online agreement here with a click through](#). Alternatively, for contributions from corporations, or those that do not wish to create an Mbed account, a slightly different agreement can be found [here](#). This agreement should be signed and returned to Arm as described in the instructions given.

Coding Standards

- We would ask that contributions conform to [our coding standards](#), and that contributions are fully tested before submission, as mentioned in the [Tests](#) and [Continuous Integration](#) sections.
- The code should be written in a clean and readable style.
- The code should be written in a portable generic way, that will benefit the whole community, and not only your own needs.
- The code should be secure, and will be reviewed from a security point of view as well.

Making a Contribution

1. [Check for open issues](#) or [start a discussion](#) around a feature idea or a bug.
2. Fork the [Mbed TLS repository on GitHub](#) to start making your changes. As a general rule, you should use the "[development](#)" branch as a basis.
3. Write a test which shows that the bug was fixed or that the feature works as expected.
4. Send a pull request (PR) and work with us until it gets merged and published. Contributions may need some modifications, so a few rounds of review and fixing may be necessary. We will include your name in the ChangeLog 😊
5. For quick merging, the contribution should be short, and concentrated on a single feature or topic. The larger the contribution is, the longer it would take to review it and merge it.
6. Mbed TLS is released under the Apache license, and as such, all the added files should include the Apache license header.

API/ABI Compatibility

The project aims to minimise the impact on users upgrading to newer versions of the library and it should not be necessary for a user to make any changes to their own code to work with a newer version of the library. Unless the user has made an active decision to use newer features, a newer generation of the library or a change has been necessary due to a security issue or other significant software defect, no modifications to their own code should be necessary. To achieve this, API compatibility is maintained between different versions of Mbed TLS on the main development branch and in LTS (Long Term Support) branches.

To minimise such disruption to users, where a change to the interface is required, all changes to the ABI or API, even on the main development branch where new features are added, need to be justifiable by either being a significant enhancement, new feature or bug fix which is best resolved by an interface change.

Where changes to an existing interface are necessary, functions in the public interface which need to be changed, are marked as 'deprecated'. This is done with the preprocessor symbols `MBEDTLS_DEPRECATED_WARNING` and `MBEDTLS_DEPRECATED_REMOVED`. Then, a new function with a new name but similar if not identical behaviour to the original function containing the necessary changes should be created alongside the existing deprecated function.

When a build is made with the deprecation preprocessor symbols defined, a compiler warning will be generated to warn a user that the function will be removed at some point in the future, notifying users that they should change from the older deprecated function to the newer function at their own convenience.

Therefore, no changes are permitted to the definition of functions in the public interface which will change the API. Instead the interface can only be changed by its extension. As described above, if a function needs to be changed, a new function needs to be created alongside it, with a new name, and whatever change is necessary, such as a new parameter or the addition of a return value.

Periodically, the library will remove deprecated functions from the library which will be a breaking change in the API, but such changes will be made only in a planned, structured way that gives sufficient notice to users of the library.

Long Term Support Branches

Mbed TLS maintains several LTS (Long Term Support) branches, which are maintained continuously for a given period. The LTS branches are provided to allow users of the library to have a maintained, stable version of the library which contains only security fixes and fixes for other defects, without encountering additional features or API extensions which may introduce issues or change the code size or RAM usage, which can be significant considerations on some platforms. To allow users to take advantage of the LTS branches, these branches maintain backwards compatibility for both the public API and ABI.

When backporting to these branches please observe the following rules:

1. Any change to the library which changes the API or ABI cannot be backported.
2. All bug fixes that correct a defect that is also present in an LTS branch must be backported to that LTS branch. If a bug fix introduces a change to the API such as a new function, the fix should be reworked to avoid the API change. API changes without very strong justification are unlikely to be accepted.
3. If a contribution is a new feature or enhancement, no backporting is required. Exceptions to this may be additional test cases or quality improvements such as changes to build or test scripts.

It would be highly appreciated if contributions are backported to LTS branches in addition to the [development branch](#) by contributors.

Currently maintained LTS branches are:

1. [mbedtls-2.7](#)
2. [mbedtls-2.16](#)

Tests

As mentioned, tests that show the correctness of the feature or bug fix should be added to the pull request, if no such tests exist.

Mbed TLS includes a comprehensive set of test suites in the `tests/` directory that are dynamically generated to produce the actual test source files (e.g. `test_suite_mpi.c`). These files are generated from a `function file` (e.g. `suites/test_suite_mpi.function`) and a `data file` (e.g. `suites/test_suite_mpi.data`). The function file contains the test functions. The data file contains the test cases, specified as parameters that will be passed to the test function.

A [Knowledge Base article describing how to add additional tests is available on the Mbed TLS website.](#)

A test script `tests/scripts/basic-build-test.sh` is available to show test coverage of the library. New code contributions should provide a similar level of code coverage to that which already exists for the library.

Sample applications, if needed, should be modified as well.

Continuous Integration Tests

Once a PR has been made, the Continuous Integration (CI) tests are triggered and run. You should follow the result of the CI tests, and fix failures.

It is advised to enable the [githooks scripts](#) prior to pushing your changes, for catching some of the issues as early as possible.

Documentation

Mbed TLS is well documented, but if you think documentation is needed, speak out!

1. All interfaces should be documented through Doxygen. New APIs should introduce Doxygen documentation.
2. Complex parts in the code should include comments.
3. If needed, a Readme file is advised.
4. If a [Knowledge Base \(KB\)](#) article should be added, write this as a comment in the PR description.
5. A [ChangeLog](#) entry should be added for this contribution.

README for Mbed TLS

Configuration

Mbed TLS should build out of the box on most systems. Some platform specific options are available in the fully documented configuration file `include/mbedtls/config.h`, which is also the place where features can be selected. This file can be edited manually, or in a more

programmatic way using the Perl script `scripts/config.pl` (use `--help` for usage instructions).

Compiler options can be set using conventional environment variables such as `CC` and `CFLAGS` when using the Make and CMake build system (see below).

Compiling

There are currently three active build systems used within Mbed TLS releases:

- GNU Make
- CMake
- Microsoft Visual Studio (Microsoft Visual Studio 2010 or later)

The main systems used for development are CMake and GNU Make. Those systems are always complete and up-to-date. The others should reflect all changes present in the CMake and Make build system, although features may not be ported there automatically.

The Make and CMake build systems create three libraries: `libmbedtls`, `libmbedx509`, and `libmbeddtls`. Note that `libmbeddtls` depends on `libmbedx509` and `libmbedcrypto`, and `libmbedx509` depends on `libmbedcrypto`. As a result, some linkers will expect flags to be in a specific order, for example the GNU linker wants `-Lmbeddtls -Lmbedx509 -Lmbedcrypto`. Also, when loading shared libraries using `dlopen()`, you'll need to load `libmbedcrypto` first, then `libmbedx509`, before you can load `libmbeddtls`.

Make

We require GNU Make. To build the library and the sample programs, GNU Make and a C compiler are sufficient. Some of the more advanced build targets require some Unix/Linux tools.

We intentionally only use a minimum of functionality in the makefiles in order to keep them as simple and independent of different toolchains as possible, to allow users to more easily move between different platforms. Users who need more features are recommended to use CMake.

In order to build from the source code using GNU Make, just enter at the command line:

```
make
```

In order to run the tests, enter:

```
make check
```

The tests need Python to be built and Perl to be run. If you don't have one of them installed, you can skip building the tests with:

```
make no_test
```

You'll still be able to run a much smaller set of tests with:

```
programs/test/selftest
```

In order to build for a Windows platform, you should use `WINDOWS_BUILD=1` if the target is Windows but the build environment is Unix-like (for instance when cross-compiling, or compiling from an MSYS shell), and `WINDOWS=1` if the build environment is a Windows shell (for instance using mingw32-make) (in that case some targets will not be available).

Setting the variable `SHARED` in your environment will build shared libraries in addition to the static libraries. Setting `DEBUG` gives you a debug build. You can override `CFLAGS` and `LDFLAGS` by setting them in your environment or on the make command line; compiler warning options may be overridden separately using `WARNING_CFLAGS`. Some directory-specific options (for example, `-I` directives) are still preserved.

Please note that setting `CFLAGS` overrides its default value of `-O2` and setting `WARNING_CFLAGS` overrides its default value (starting with `-Wall -W`), so if you just want to add some warning options to the default ones, you can do so by setting `CFLAGS=-O2 -Werror` for example.

Setting `WARNING_CFLAGS` is useful when you want to get rid of its default content (for example because your compiler doesn't accept `-Wai` as an option). Directory-specific options cannot be overridden from the command line.

Depending on your platform, you might run into some issues. Please check the Makefiles in `library/`, `programs/` and `tests/` for options to manually add or remove for specific platforms. You can also check [the Mbed TLS Knowledge Base](#) for articles on your platform or issue.

In case you find that you need to do something else as well, please let us know what, so we can add it to the [Mbed TLS Knowledge Base](#).

CMake

In order to build the source using CMake in a separate directory (recommended), just enter at the command line:

```
mkdir /path/to/build_dir && cd /path/to/build_dir  
cmake /path/to/mbedtls_source  
make
```

In order to run the tests, enter:

```
make test
```

The test suites need Python to be built and Perl to be executed. If you don't have one of these installed, you'll want to disable the test suites with:

```
cmake -DENABLE_TESTING=Off /path/to/mbedtls_source
```

If you disabled the test suites, but kept the programs enabled, you can still run a much smaller set of tests with:

```
programs/test/selftest
```

To configure CMake for building shared libraries, use:

```
cmake -DUSE_SHARED_MBEDTLS_LIBRARY=On /path/to/mbedtls_source
```

There are many different build modes available within the CMake buildsystem. Most of them are available for gcc and clang, though some are compiler-specific:

- `Release`. This generates the default code without any unnecessary information in the binary files.
- `Debug`. This generates debug information and disables optimization of the code.
- `Coverage`. This generates code coverage information in addition to debug information.
- `ASan`. This instruments the code with AddressSanitizer to check for memory errors. (This includes LeakSanitizer, with recent version of gcc and clang.) (With recent version of clang, this mode also instruments the code with UndefinedSanitizer to check for undefined behaviour.)
- `ASanDbg`. Same as ASan but slower, with debug information and better stack traces.
- `MemSan`. This instruments the code with MemorySanitizer to check for uninitialized memory reads. Experimental, needs recent clang on Linux/x86_64.
- `MemSanDbg`. Same as MemSan but slower, with debug information, better stack traces and origin tracking.
- `Check`. This activates the compiler warnings that depend on optimization and treats all warnings as errors.

Switching build modes in CMake is simple. For debug mode, enter at the command line:

```
cmake -D CMAKE_BUILD_TYPE=Debug /path/to/mbedtls_source
```

To list other available CMake options, use:

```
cmake -LH
```

Note that, with CMake, you can't adjust the compiler or its flags after the

initial invocation of cmake. This means that `CC=your_cc` make and `make CC=your_cc` will not work (similarly with `CFLAGS` and other variables).

These variables need to be adjusted when invoking cmake for the first time, for example:

```
CC=your_cc cmake /path/to/mbedtls_source
```

If you already invoked cmake and want to change those settings, you need to remove the build directory and create it again.

Note that it is possible to build in-place; this will however overwrite the provided Makefiles (see `scripts/tmp_ignore_makefiles.sh` if you want to prevent `git status` from showing them as modified). In order to do so, from the Mbed TLS source directory, use:

```
cmake .
make
```

If you want to change `CC` or `CFLAGS` afterwards, you will need to remove the CMake cache. This can be done with the following command using GNU find:

```
find . -iname '*cmake*' -not -name CMakeLists.txt -exec rm -rf {} +
```

You can now make the desired change:

```
CC=your_cc cmake .
make
```

Regarding variables, also note that if you set `CFLAGS` when invoking `cmake`, your value of `CFLAGS` doesn't override the content provided by `cmake` (depending on the build mode as seen above), it's merely prepended to it.

Microsoft Visual Studio

The build files for Microsoft Visual Studio are generated for Visual Studio 2010.

The solution file `mbedtls.sln` contains all the basic projects needed to build the library and all the programs. The files in tests are not generated and compiled, as these need Python and perl environments as well. However, the selftest program in `programs/test/` is still available.

Example programs

We've included example programs for a lot of different features and uses in `programs/`. Most programs only focus on a single feature or usage scenario, so keep that in mind when copying parts of the code.

Tests

Mbed TLS includes an elaborate test suite in `tests/` that initially requires Python to generate the tests files (e.g. `test_suite\mpi.c`). These files are generated from a `function file` (e.g. `suites/test_suite\mpi.function`) and a `data file` (e.g. `suites/test_suite\mpi.data`). The `function file` contains the test functions. The `data file` contains the test cases, specified as parameters that will be passed to the test function.

For machines with a Unix shell and OpenSSL (and optionally GnuTLS) installed, additional test scripts are available:

- `tests/ssl-opt.sh` runs integration tests for various TLS options (renegotiation, resumption, etc.) and tests interoperability of these options with other implementations.
- `tests/compat.sh` tests interoperability of every ciphersuite with other implementations.

- `tests/scripts/test-ref-configs.pl` test builds in various reduced configurations.
- `tests/scripts/key-exchanges.pl` test builds in configurations with a single key exchange enabled
- `tests/scripts/all.sh` runs a combination of the above tests, plus some more, with various build options (such as ASan, full `config.h`, etc).

Configurations

We provide some non-standard configurations focused on specific use cases in the `configs/` directory. You can read more about those in `configs/README.txt`

Porting Mbed TLS

Mbed TLS can be ported to many different architectures, OS's and platforms. Before starting a port, you may find the following Knowledge Base articles useful:

- [Porting Mbed TLS to a new environment or OS](#)
- [What external dependencies does Mbed TLS rely on?](#)
- [How do I configure Mbed TLS](#)

Contributing

We gratefully accept bug reports and contributions from the community. There are some requirements we need to fulfill in order to be able to integrate contributions:

- All contributions, whether large or small require a Contributor's License Agreement (CLA) to be accepted. This is because source code can possibly fall under copyright law and we need your consent to share in the ownership of the copyright.
- We would ask that contributions conform to [our coding standards](#), and that contributions should be fully tested before submission.
- As with any open source project, contributions will be reviewed by the project team and community and may need some modifications to be accepted.

To accept the Contributor's Licence Agreement (CLA), individual contributors can do this by creating an Mbed account and [accepting the online agreement here with a click through](#). Alternatively, for contributions from corporations, or those that do not wish to create an Mbed account, a slightly different agreement can be found [here](#). This agreement should be signed and returned to Arm as described in the instructions given.

Making a Contribution

1. [Check for open issues](#) or [start a discussion](#) around a feature idea or a bug.
2. Fork the [Mbed TLS repository on GitHub](#) to start making your changes. As a general rule, you should use the "development" branch as a basis.
3. Write a test which shows that the bug was fixed or that the feature works as expected.
4. Send a pull request and bug us until it gets merged and published. Contributions may need some modifications, so work with us to get your change accepted. We will include your name in the ChangeLog 😊

Mbed TLS sample programs

This subdirectory mostly contains sample programs that illustrate specific features of the library, as well as a few test and support programs.

Symmetric cryptography (AES) examples

- `aes/aescrypt2.c`: file encryption and authentication with a key derived from a low-entropy secret, demonstrating the low-level AES interface, the digest interface and HMAC.

Warning: this program illustrates how to use low-level functions in the library. It should not be taken as an example of how to build a secure encryption mechanism. To derive a key from a low-entropy secret such as a password, use a standard key stretching mechanism such as PBKDF2 (provided by the `pkcs5` module). To encrypt and authenticate data, use a standard mode such as GCM or CCM (both available as library module).

- `aes/crypt_and_hash.c`: file encryption and authentication, demonstrating the generic cipher interface and the generic hash interface.

Hash (digest) examples

- `hash/generic_sum.c`: file hash calculator and verifier, demonstrating the message digest (`md`) interface.
- `hash/hello.c`: hello-world program for MD5.

Public-key cryptography examples

Generic public-key cryptography (`pk`) examples

- `pkey/gen_key.c`: generates a key for any of the supported public-key algorithms (RSA or ECC) and writes it to a file that can be used by the other pk sample programs.
- `pkey/key_app.c`: loads a PEM or DER public key or private key file and dumps its content.
- `pkey/key_app_writer.c`: loads a PEM or DER public key or private key file and writes it to a new PEM or DER file.
- `pkey/pk_encrypt.c`, `pkey/pk_decrypt.c`: loads a PEM or DER public/private key file and uses the key to encrypt/decrypt a short string through the generic public-key interface.
- `pkey/pk_sign.c`, `pkey/pk_verify.c`: loads a PEM or DER private/public key file and uses the key to sign/verify a short string.

ECDSA and RSA signature examples

- `pkey/ecdsa.c`: generates an ECDSA key, signs a fixed message and verifies the signature.
- `pkey/rsa_encrypt.c`, `pkey/rsa_decrypt.c`: loads an RSA public/private key and uses it to encrypt/decrypt a short string through the low-level RSA interface.
- `pkey/rsa_genkey.c`: generates an RSA key and writes it to a file that can be used with the other RSA sample programs.
- `pkey/rsa_sign.c`, `pkey/rsa_verify.c`: loads an RSA private/public key and uses it to sign/verify a short string with the RSA PKCS#1 v1.5 algorithm.
- `pkey/rsa_sign_pss.c`, `pkey/rsa_verify_pss.c`: loads an RSA private/public key and uses it to sign/verify a short string with the RSASSA-PSS algorithm.

Diffie-Hellman key exchange examples

- `pkey/dh_client.c`, `pkey/dh_server.c`: secure channel demonstrators (client, server). This pair of programs illustrates how to set up a secure channel using RSA for authentication and Diffie-Hellman to generate a shared AES session key.
- `pkey/ecdh_curve25519.c`: demonstration of a elliptic curve Diffie-Hellman (ECDH) key agreement.

Bignum (`mpi`) usage examples

- `pkey/dh_genprime.c`: shows how to use the bignum (`mpi`) interface to generate Diffie-Hellman parameters.
- `pkey/mpi_demo.c`: demonstrates operations on big integers.

Random number generator (RNG) examples

- `random/gen_entropy.c`: shows how to use the default entropy sources to generate random data.
Note: most applications should only use the entropy generator to seed a cryptographic pseudorandom generator, as illustrated by `random/gen_random_ctr_drbg.c`.
- `random/gen_random_ctr_drbg.c`: shows how to use the default entropy sources to seed a pseudorandom generator, and how to use the resulting random generator to generate random data.
- `random/gen_random_havege.c`: demonstrates the HAVEGE entropy collector.

SSL/TLS examples

SSL/TLS sample applications

- `ssl/dtls_client.c`: a simple DTLS client program, which sends one datagram to the server and reads one datagram in response.
- `ssl/dtls_server.c`: a simple DTLS server program, which expects one datagram from the client and writes one datagram in response. This program supports DTLS cookies for hello verification.
- `ssl/mini_client.c`: a minimalistic SSL client, which sends a short string and disconnects. This is primarily intended as a benchmark; for a better example of a typical TLS client, see `ssl/ssl_client1.c`.
- `ssl/ssl_client1.c`: a simple HTTPS client that sends a fixed request and displays the response.
- `ssl/ssl_fork_server.c`: a simple HTTPS server using one process per client to send a fixed response. This program requires a Unix/POSIX environment implementing the `fork` system call.
- `ssl/ssl_mail_client.c`: a simple SMTP-over-TLS or SMTP-STARTTLS client. This client sends an email with fixed content.
- `ssl/ssl_pthread_server.c`: a simple HTTPS server using one thread per client to send a fixed response. This program requires the `pthread` library.
- `ssl/ssl_server.c`: a simple HTTPS server that sends a fixed response. It serves a single client at a time.

SSL/TLS feature demonstrators

Note: unlike most of the other programs under the `programs/` directory, these two programs are not intended as a basis for writing an application. They combine most of the features supported by the library, and most applications require only a few features. To write a new application, we recommended that you start with `ssl_client1.c` or `ssl_server.c`, and then look inside `ssl/ssl_client2.c` or `ssl/ssl_server2.c` to see how to use the specific features that your application needs.

- `ssl/ssl_client2.c`: an HTTPS client that sends a fixed request and displays the response, with options to select TLS protocol features and Mbed TLS library features.
- `ssl/ssl_server2.c`: an HTTPS server that sends a fixed response, with options to select TLS protocol features and Mbed TLS library features.

In addition to providing options for testing client-side features, the `ssl_client2` program has options that allow you to trigger certain behaviors in the server. For example, there are options to select ciphersuites, or to force a renegotiation. These options are useful for testing the corresponding features in a TLS server. Likewise, `ssl_server2` has options to activate certain behaviors that are useful for testing a TLS client.

Test utilities

- `test/benchmark.c`: benchmark for cryptographic algorithms.
- `test/selftest.c`: runs the self-test function in each library module.
- `test/udp_proxy.c`: a UDP proxy that can inject certain failures (delay, duplicate, drop). Useful for testing DTLS.
- `test/zeroize.c`: a test program for `mbedtls_platform_zeroize`, used by `tests/scripts/test_zeroize.gdb`.

Development utilities

- `util/pem2der.c`: a PEM to DER converter. Mbed TLS can read PEM files directly, but this utility can be useful for interacting with other tools or with minimal Mbed TLS builds that lack PEM support.
- `util/strerror.c`: prints the error description corresponding to an integer status returned by an Mbed TLS function.

X.509 certificate examples

- `x509/cert_app.c`: connects to a TLS server and verifies its certificate chain.
- `x509/cert_req.c`: generates a certificate signing request (CSR) for a private key.
- `x509/cert_write.c`: signs a certificate signing request, or self-signs a certificate.
- `x509/crl_app.c`: loads and dumps a certificate revocation list (CRL).
- `x509/req_app.c`: loads and dumps a certificate signing request (CSR).

mbedTLS library for Mongoose OS

This library is optimized by Cesanta for lower memory usage.

It also includes support for ATECC508 crypto chip.

GATT Client RPC service

Provides [RPC service](#) for

[GATT](#)

client.

Methods

- `GATTC.Scan` - performs BLE scan of the area. Results are returned as an array of objects containing address, name (if present) and RSSI.

```
$ mos call GATTC.Scan
{
  "results": [
    {
      "addr": "eb:12:dd:51:19:3d",
      "rssi": -48
    },
    {
      "addr": "24:0a:c4:00:31:be",
      "name": "esp32_0031BC",
      "rssi": -36
    }
  ]
}
```

- `GATTC.Open` - open connection to a device. Device can be specified by either `addr` or `name`. Optional `mtu` parameter specifies link MTU to be used, the default is 23 bytes (as per standard). Returned value is `conn_id` which is the connection identifier to be used for later operations.

```
$ mos call GATTC.Open '{"addr": "eb:12:dd:51:19:3d"}'
{
  "conn_id": 0
}
$ mos call GATTC.Open '{"name": "esp32_0031BC", "mtu": 200}'
{
  "conn_id": 1
}
```

- `GATTS.ListServices` - list services provided by device. For each discovered service, its UUID, instance number and primary flag are returned.

```
$ mos call GATTC.ListServices '{"conn_id": 1}'
{
  "results": [
    {
      "uuid": "1801",
      "instance": 0,
      "primary": true
    },
    {
      "uuid": "1800",
      "instance": 0,
      "primary": true
    },
    {
      "uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f",
      "instance": 0,
      "primary": true
    },
    {
      "uuid": "5f6d4f53-5f52-5043-5f53-56435f49445f",
      "instance": 0,
      "primary": true
    }
  ]
}
```

In this example, we see two standard services - [Generic Access](#) (1800) and [Generic Attribute](#), followed by custom mOS [configuration](#) and [RPC over GATT](#) services.

- [GATTS.ListCharacteristics](#) - list characteristics of a service. For each characteristic, its UUID and properties are returned.

Properties are returned a string of up to 8 characters:

- R - read
- W - write
- w - write with no response required
- N - notify
- I - indicate
- B - broadcast
- E - extended
- A - auth

```
$ mos call GATTC.ListCharacteristics '{"conn_id": 1, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f"}'
{
  "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f",
  "results": [
    {
      "uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30",
      "props": "W"
    },
    {
      "uuid": "316d4f53-5f43-4647-5f76-616c75655f31",
      "props": "WR"
    },
    {
      "uuid": "326d4f53-5f43-4647-5f73-6176655f5f32",
      "props": "W"
    }
  ]
}
```

- [GATTC.Write](#) - write a value to the specified characteristic. `conn_id`, `svc_uuid` and `char_uuid` specify the characteristic, value can be provided either as plain text `value` or hex-encoded `value_hex` keys. The following two calls are equivalent

```
$ mos call GATT.C.Write '{"conn_id": 1, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30", "value": null}
$ mos call GATT.C.Write '{"conn_id": 1, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30", "value": null}
```

- GATT.C.Read - read avlue of a characteristic `conn_id`, `svc_uuid` and `char_uuid` specify the characteristic, value will be returned as either `value` or `value_hex`: if value is a string consisting of printable charaters that can be represented as a valid JSON string, result will be in the `value` key. Otherwise it will be hexified and sent in `value_hex`.

```
$ mos call GATT.C.Read '{"conn_id": 1, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "316d4f53-5f43-4647-5f76-616c75655f31"}'
{
  "value": "true"
}
```

- GATT.C.Close - close the specified connection.

```
$ mos call GATT.C.Close '{"conn_id": 1}'
null
```

Provisioning one device using another

As a complete example, here is full Bluetooth provisioning process described in [this blog post](#):

```
$ mos call GATT.C.Open '{"name": "esp32_0031BC", "mtu": 200}'
{
  "conn_id": 0
}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "316d4f53-5f43-4647-5f76-616c75655f31", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "326d4f53-5f43-4647-5f73-6176655f5f32", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "316d4f53-5f43-4647-5f76-616c75655f31", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "326d4f53-5f43-4647-5f73-6176655f5f32", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "306d4f53-5f43-4647-5f6b-65795f5f5f30", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "316d4f53-5f43-4647-5f76-616c75655f31", "value": null}
$ mos call GATT.C.Write '{"conn_id": 0, "svc_uuid": "5f6d4f53-5f43-4647-5f53-56435f49445f", "char_uuid": "326d4f53-5f43-4647-5f73-6176655f5f32", "value": null}
```

Arduino DS3231 High Precision Real Time Clock (RTC) library for Mongoose OS

Tested and works on esp8266/esp32

Description

DS3231 is a low-cost, extremely accurate I²C real-time clock (RTC), with an integrated temperature-compensated crystal oscillator (TCXO) and crystal. The device incorporates a battery input, disconnect the main power supply and maintains accurate timekeeping. Integrated oscillator improve long-term accuracy of the device and reduces the number of components of the production line.

RTC maintains seconds, minutes, hours, day, date, month, and year information. Less than 31 days of the month, the end date will be automatically adjusted, including corrections for leap year. The clock operates in either the 24 hours or band / AM / PM indication of the 12-hour format. Provides two configurable alarm clock and a calendar can be set to a square wave output. Address and data are transferred serially through an I2C bidirectional bus. See [product page link](#) for more information about the hardware.

Features

- Battery
- Operating voltage: 3.3 - 5.5V
- Clock chip: High-precision clock chip DS3231
- Clock Accuracy: 0-40 °C range, the accuracy 2ppm, the error was about 1 minute
- Calendar alarm clock with two
- Programmable square-wave output
- Real time clock generator seconds, minutes, hours, day, date, month and year timing and provide valid until the year 2100 leap year compensation

Usage

mos.yml, add:

```
config_schema:  
- ["i2c.enable", true]  
libs:  
- origin: https://github.com/mongoose-os-libs/arduino-ds3231
```

init.js, add:

```
load('api_ds3231.js');
```

[\(JavaScript\) usage example](#)

```
# include "mgos_arduino_DS3231.h"  
<p class="mume-header" id="include-mgos_arduino_ds3231h"></p>
```

How to contribute

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

SHT31 I2C Driver

A Mongoose library for Sensirion's SHT31 integrated circuit.

Sensor details

SHT3x-DIS is the next generation of Sensirion's temperature and humidity sensors. It builds on a new CMOSens sensor chip that is at the heart of Sensirion's new humidity and temperature platform. The SHT3x-DIS has increased intelligence, reliability and improved accuracy specifications compared to its predecessor. Its functionality includes enhanced signal processing, two distinctive and user selectable I2C addresses and communication speeds of up to 1 MHz. The DFN package has a footprint of 2.5 x 2.5 mm while keeping a height of 0.9 mm. This allows for integration of the SHT3x-DIS into a great variety of applications.

Additionally, the wide supply voltage range of 2.4 to 5.5 V guarantees compatibility with diverse assembly situations. All in all, the SHT3x-DIS incorporates 15 years of knowledge of Sensirion, the leader in the humidity sensor industry.

See [datasheet](#)

for implementation details.

A great place to buy a ready made and tested unit is at [Adafruit](#).

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```
# include "mgos.h"
<p class="mume-header" id="include-mgosh-25"></p>

# include "mgos_i2c.h"
<p class="mume-header" id="include-mgos_i2ch-4"></p>

# include "mgos_sht31.h"
<p class="mume-header" id="include-mgos_sht31h"></p>

static struct mgos_sht31 *s_sht31;

static void timer_cb(void *user_data) {
    float temperature, humidity;

    temperature=mgos_sht31_getTemperature(s_sht31);
    humidity=mgos_sht31_getHumidity(s_sht31);

    LOG(LL_INFO, ("sht31 temperature=%.2f humidity=%.2f", temperature, humidity));

    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_i2c *i2c;

    i2c=mgos_i2c_get_global();
    if (!i2c) {
        LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
    } else {
        s_sht31=mgos_sht31_create(i2c, 0x40); // Default I2C address
        if (s_sht31) {
            mgos_set_timer(5000, true, timer_cb, NULL);
        } else {
            LOG(LL_ERROR, ("Could not initialize sensor"));
        }
    }
    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

SPI Flash VFS Driver

This driver supports external SPI flash chips and makes them available for use with the mOS VFS subsystem.

It add an `spi_flash` device type.

Supported options (`dev_opts`) are:

- `freq` - SPI bus frequency.
- `cs` - SPI bus CS line to use, see the [SPI example](#) for explanation.
- `mode` - SPI mode, 0 - 2.

- `size` - specify size explicitly. If not specified, an attempt is made to detect by querying SDFP data or JEDEC ID. Most modern chips support at least one of these, so specifying size is usually not necessary.
- `wip_mask` - bit mask to apply to status register to determine when the write is in progress (chip is busy). Most chips have bit 0 as the WIP bit, and the corresponding mask value is 1. This is the default.

Example of console output when this driver is used:

```
[Jul 28 14:35:20.697] mgos_vfs_dev_open spi_flash ({ "freq": 80000000, "cs": 0 }) -> 0x3ffb47f0
[Jul 28 14:35:20.704] mgos_vfs_dev_spi_fla Chip ID: 20 71, size: 1048576
[Jul 28 14:35:20.714] mgos_vfs_mkfs Create SPIFFS (dev 0x3ffb47f0, opts {"size": 262144, "force": true})
```

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent

to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution." "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and

do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions.

Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Apple HomeKit ADK for Mongoose OS

This is a port of the official [Apple HomeKit ADK](#) to Mongoose OS.

Targeted platforms are ESP8266 and ESP32 for now.

Examples

Example Mongoose OS app that uses this library can be found [here](#).

TODO

- MFi chip support
- BLE transport support on ESP32
- Buffer size optimization (<https://github.com/apple/HomeKitADK/issues/13>).

HomeKit ADK Platform Abstraction Layer (PAL)

This document describes the ADK PAL and how to port the ADK to a new platform.

Supported platforms

The ADK ships with existing implementations for the following platforms and cryptographic libraries:

Operating systems and runtime environments

- Darwin (macOS)

- Linux

Cryptographic libraries

- OpenSSL (1.1.1c or later)
- MbedTLS (2.18.0 or later)

Additional platforms can be supported by providing a custom PAL implementation.

Design considerations

The PAL abstracts all relevant aspects of the underlying platform. Currently the PAL attempts to avoid memory allocations. This is not necessarily a design goal we'll maintain going forward. The PAL has 3 build types: Test, Debug, and Release. Test uses the Mock PAL and is used for unit tests. Debug and Release use platform-specific backends.

Configuration and weak symbols

Past versions of the ADK used platform-specific constructors to pass options to PAL constructors. The current version of the ADK instead uses a consistent constructor across all platforms. Our goal is for applications to be portable across platforms and across BLE and IP.

Crypto PAL

Ed25519

EdDSA (Edwards-curve Digital Signature Algorithm) signature scheme using SHA-512 and Curve25519.

MbedTLS does not natively support Ed25519. We ship a performant software-only implementation along with our MbedTLS implementation but we strongly encourage silicon vendors to substitute their own production quality implementation (or hardware accelerated implementation).

To disable our implementation we recommend including the C file of our bindings in a new C file and defining `HAVE_CUSTOM_ED25519`, which disables our implementation.

```
# define ED25519_PUBLIC_KEY_BYTES 32
<p class="mume-header" id="define-ed25519_public_key_bytes-32"></p>

# define ED25519_SECRET_KEY_BYTES 32
<p class="mume-header" id="define-ed25519_secret_key_bytes-32"></p>

# define ED25519_BYTES 64
<p class="mume-header" id="define-ed25519_bytes-64"></p>

void HAP_ed25519_public_key(uint8_t pk[ED25519_PUBLIC_KEY_BYTES],
                           const uint8_t sk[ED25519_SECRET_KEY_BYTES]);
void HAP_ed25519_sign(uint8_t sig[ED25519_BYTES],
                      const uint8_t *m, size_t m_len,
                      const uint8_t sk[ED25519_SECRET_KEY_BYTES],
                      const uint8_t pk[ED25519_PUBLIC_KEY_BYTES]);
int HAP_ed25519_verify(const uint8_t sig[ED25519_BYTES],
                       const uint8_t *m, size_t m_len,
                       const uint8_t pk[ED25519_PUBLIC_KEY_BYTES]);
```

X25519

X25519 is an elliptic curve Diffie-Hellman key exchange scheme using Curve25519.

```
# define X25519_SCALAR_BYTES 32
<p class="mume-header " id="define-x25519_scalar_bytes-32"></p>

# define X25519_BYTES 32
<p class="mume-header " id="define-x25519_bytes-32"></p>

void HAP_X25519_scalarmult_base(uint8_t r[X25519_BYTES],
                                const uint8_t n[X25519_SCALAR_BYTES]);
void HAP_X25519_scalarmult(uint8_t r[X25519_BYTES],
                           const uint8_t n[X25519_SCALAR_BYTES],
                           const uint8_t p[X25519_BYTES]);
```

ChaCha20 Poly1305

Authenticated Encryption using ChaCha20 stream cipher and Poly1305 authenticator.

```
# define CHACHA20_POLY1305_KEY_BYTES 32
<p class="mume-header " id="define-chacha20_poly1305_key_bytes-32"></p>

# define CHACHA20_POLY1305_NONCE_BYTES_MAX 12
<p class="mume-header " id="define-chacha20_poly1305_nonce_bytes_max-12"></p>

# define CHACHA20_POLY1305_TAG_BYTES 16
<p class="mume-header " id="define-chacha20_poly1305_tag_bytes-16"></p>

void HAP_chacha20_poly1305_init(HAP_chacha20_poly1305_ctx *ctx,
                                  const uint8_t *n, size_t n_len,
                                  const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
void HAP_chacha20_poly1305_update_enc(HAP_chacha20_poly1305_ctx *ctx,
                                       uint8_t *c,
                                       const uint8_t *m, size_t m_len,
                                       const uint8_t *n, size_t n_len,
                                       const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
void HAP_chacha20_poly1305_update_enc_aad(HAP_chacha20_poly1305_ctx *ctx,
                                            const uint8_t *a, size_t a_len,
                                            const uint8_t *n, size_t n_len,
                                            const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
void HAP_chacha20_poly1305_final_enc(HAP_chacha20_poly1305_ctx *ctx,
                                      uint8_t tag[CHACHA20_POLY1305_TAG_BYTES]);
void HAP_chacha20_poly1305_update_dec(HAP_chacha20_poly1305_ctx *ctx,
                                       uint8_t *m,
                                       const uint8_t *c, size_t c_len,
                                       const uint8_t *n, size_t n_len,
                                       const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
void HAP_chacha20_poly1305_update_dec_aad(HAP_chacha20_poly1305_ctx *ctx,
                                            const uint8_t *a, size_t a_len,
                                            const uint8_t *n, size_t n_len,
                                            const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
int HAP_chacha20_poly1305_final_dec(HAP_chacha20_poly1305_ctx *ctx,
                                     const uint8_t tag[CHACHA20_POLY1305_TAG_BYTES]);
```

Note: *The implementation must support overlapping buffers (m and c).*

If HAVE_CUSTOM_SINGLE_SHOT_CHACHA20_POLY1305 is not set, we synthesize a single shot API from the streaming API above. Otherwise the backend must provide the following API. This is recommended for BLE-only crypto backends. *The streaming API is needed by IP accessories only.*

```
void HAP_chacha20_poly1305_encrypt_aad(uint8_t tag[CHACHA20_POLY1305_TAG_BYTES],
    uint8_t *c,
    const uint8_t *m, size_t m_len,
    const uint8_t *a, size_t a_len,
    const uint8_t *n, size_t n_len,
    const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);

int HAP_chacha20_poly1305_decrypt_aad(const uint8_t tag[CHACHA20_POLY1305_TAG_BYTES],
    uint8_t *m,
    const uint8_t *c, size_t c_len,
    const uint8_t *a, size_t a_len,
    const uint8_t *n, size_t n_len,
    const uint8_t k[CHACHA20_POLY1305_KEY_BYTES]);
```

Note: a_len might be NULL.

SRP6a

Secure Remote Password protocol (SRP6a), an augmented password-authenticated key agreement (PAKE) protocol.

Both OpenSSL and MbedTLS do not natively support SRP6a. We are shipping an SRP6a implementation with our code that uses the BIGNUM support in OpenSSL and MbedTLS respectively. We strongly recommend substituting an optimized implementation for MCU-class silicon.

To replace our implementation we recommend including the C file of our bindings in a new C file and defining *CUSTOM_SRP*, which disables our implementation.

```

#define SRP_PRIME_BYTES 384
<p class="mume-header " id="define-srp_prime_bytes-384"></p>

#define SRP_SALT_BYTES 16
<p class="mume-header " id="define-srp_salt_bytes-16"></p>

#define SRP_VERIFIER_BYTES 384
<p class="mume-header " id="define-srp_verifier_bytes-384"></p>

#define SRP_SECRET_KEY_BYTES 32
<p class="mume-header " id="define-srp_secret_key_bytes-32"></p>

#define SRP_PUBLIC_KEY_BYTES 384
<p class="mume-header " id="define-srp_public_key_bytes-384"></p>

#define SRP_SCRAMBLING_PARAMETER_BYTES 64
<p class="mume-header " id="define-srp_scrambling_parameter_bytes-64"></p>

#define SRP_PREMASTER_SECRET_BYTES 384
<p class="mume-header " id="define-srp_premaster_secret_bytes-384"></p>

#define SRP_SESSION_KEY_BYTES 64
<p class="mume-header " id="define-srp_session_key_bytes-64"></p>

#define SRP_PROOF_BYTES 64
<p class="mume-header " id="define-srp_proof_bytes-64"></p>

void HAP_srp_verifier(uint8_t v[SRP_VERIFIER_BYTES],
                      const uint8_t salt[SRP_SALT_BYTES],
                      const uint8_t *user, size_t user_len,
                      const uint8_t *pass, size_t pass_len);
void HAP_srp_public_key(uint8_t pub_b[SRP_PUBLIC_KEY_BYTES],
                       const uint8_t priv_b[SRP_SECRET_KEY_BYTES],
                       const uint8_t v[SRP_VERIFIER_BYTES]);
void HAP_srp_scrambling_parameter(uint8_t u[SRP_SCRAMBLING_PARAMETER_BYTES],
                                  const uint8_t pub_a[SRP_PUBLIC_KEY_BYTES],
                                  const uint8_t pub_b[SRP_PUBLIC_KEY_BYTES]);
int HAP_srp_premaster_secret(uint8_t s[SRP_PREMASTER_SECRET_BYTES],
                            const uint8_t pub_a[SRP_PUBLIC_KEY_BYTES],
                            const uint8_t priv_b[SRP_SECRET_KEY_BYTES],
                            const uint8_t u[SRP_SCRAMBLING_PARAMETER_BYTES],
                            const uint8_t v[SRP_VERIFIER_BYTES]);
void HAP_srp_session_key(uint8_t k[SRP_SESSION_KEY_BYTES],
                        const uint8_t s[SRP_PREMASTER_SECRET_BYTES]);
void HAP_srp_proof_m1(uint8_t m1[SRP_PROOF_BYTES],
                      const uint8_t *user, size_t user_len,
                      const uint8_t salt[SRP_SALT_BYTES],
                      const uint8_t pub_a[SRP_PUBLIC_KEY_BYTES],
                      const uint8_t pub_b[SRP_PUBLIC_KEY_BYTES],
                      const uint8_t k[SRP_SESSION_KEY_BYTES]);
void HAP_srp_proof_m2(uint8_t m2[SRP_PROOF_BYTES],
                      const uint8_t pub_a[SRP_PUBLIC_KEY_BYTES],
                      const uint8_t m1[SRP_PROOF_BYTES],
                      const uint8_t k[SRP_SESSION_KEY_BYTES]);

```

SHA

Secure Hash Algorithms (SHA1, SHA-256, and SHA-512)

```
# define SHA1_BYT 20
<p class="mume-header " id="define-sha1_bytes-20"></p>

void HAP_sha1(uint8_t md[SHA1_BYT], const uint8_t *data, size_t size);

# define SHA256_BYT 32
<p class="mume-header " id="define-sha256_bytes-32"></p>

void HAP_sha256(uint8_t md[SHA256_BYT], const uint8_t *data, size_t size);

# define SHA512_BYT 64
<p class="mume-header " id="define-sha512_bytes-64"></p>

void HAP_sha512(uint8_t md[SHA512_BYT], const uint8_t *data, size_t size);
```

HMAC SHA1

Hash-based message authentication code using SHA1. *Only needed for IP Accessories.*

```
# define HMAC_SHA1_BYT SHA1_BYT
<p class="mume-header " id="define-hmac_sha1_bytes-sha1_bytes"></p>

void HAP_hmac_sha1_aad(uint8_t r[HMAC_SHA1_BYT],
    const uint8_t* key, size_t key_len,
    const uint8_t* in, size_t in_len,
    const uint8_t* aad, size_t aad_len);
```

HKDF-SHA512

HMAC-based key derivation function using SHA-512.

```
void HAP_hkdf_sha512(uint8_t* r, size_t r_len,
    const uint8_t* key, size_t key_len,
    const uint8_t* salt, size_t salt_len,
    const uint8_t* info, size_t info_len);
```

Brute-force resistant password-based key derivation function.

PBKDF2-SHA1

```
void HAP_pbkdf2_hmac_sha1(uint8_t *key, size_t key_len,
    const uint8_t *password, size_t password_len,
    const uint8_t *salt, size_t salt_len,
    uint32_t count);
```

AES-CTR

AES block cipher in CTR mode. *Only needed for IP Accessories.*

```
void HAP_aes_ctr_init(HAP_aes_ctr_ctx *ctx, const uint8_t *key, int size, const uint8_t iv[16]);
void HAP_aes_ctr_encrypt(HAP_aes_ctr_ctx *ctx, uint8_t* ct, const uint8_t* pt, size_t pt_len);
void HAP_aes_ctr_decrypt(HAP_aes_ctr_ctx *ctx, uint8_t* pt, const uint8_t* ct, size_t ct_len);
void HAP_aes_ctr_done(HAP_aes_ctr_ctx *ctx);
```

Debugging on Darwin using visual debugger

Using Visual Studio Code

Installation

- Download VSCode: <https://code.visualstudio.com/>
- To install c/c++ language support: within VSCode press ⌘+p and enter

```
ext install ms-vscode.cpptools
```

- To install debugger: within VSCode, press ⌘+p and enter

```
ext install vadimcn.vscode-lldb
```

- Note: an alternative debugger is necessary because llvm no longer includes lldb-mi which the default debugger uses.

Running an application

- Create a workspace which points to the root of your ADK folder.
- Click "Debug" > "Open Configuration", which will open the launch.json file.
- Set "type" to "lldb"
- Set "program" to the application you want to run.

Example launch.json file for Lightbulb app:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "lldb",
      "request": "launch",
      "name": "Debug",
      "program": "${workspaceFolder}/Output/Darwin-x86_64-apple-darwin19.0.0/Debug/IP/Applications/Lightbulb.OpenSSL",
      "args": [],
      "cwd": "${workspaceFolder}"
    }
  ]
}
```

Development Environment

Development of HomeKit ADK is supported on the following platforms:

Darwin

Prerequisites

Download and install [Xcode 11](#)

```
brew install openssl@1.1
brew install mbedtls --HEAD
brew install wget
brew install qemu
brew cask install docker
```

Run docker (Look in Spotlight/Applications folder). This is a one time instruction.

Make sure you go to Docker→Preferences→General and check the option → Start Docker Desktop when you log in

Linux

Prerequisites

```
sudo apt install docker
```

Code Style

Please use the following tools to auto-format your code before submitting a Pull Request.

Linting source code

This project uses `clang-format` tool to lint and format the code.

```
./Tools/linters/clint.sh -h
```

Linting shell scripts

```
./Tools/linters/shlint.sh
```

Guidelines

Sample Application

- In DB.c, the following IIDs should be used:
 - 0x...0 Service IID.
 - 0x...1 Service Signature characteristic IID. Only present when the service has linked service or has properties.
 - 0x...2 Name characteristic IID. Services can have a Name characteristic if they are user visible.
 - 0x...3+ Regular characteristics, ordered the same way as in the service specification, with required characteristics listed before optional characteristics.

When updating a DB file for a sample that has already been released, try to avoid changing existing IIDs.

Error Handling

- Unexpected errors `HAPError` are used for exceptional circumstances such as communication failure with I2C, or access problems with key-value store. Those errors should be modeled as `HAPError` and are in most cases bubbled out to the caller until the next error handler.
- Expected errors are conditions such as a key that is not available in the key-value store, or a Wi-Fi configuration that has not yet been set. Such errors should be modeled as separate out parameters, such as `bool *found`, `bool *isSet`, `bool *isDefined`, `bool *isAvailable`, or `bool *isConfigured`. Do not use `bool *isValid` to denote existence of a value.
- The idea is that a caller of a function in general should always be able to treat a `HAPError` as a boolean, and only has to handle expected errors returned through the separate out parameters.
- When a `HAPError` is returned, the caller shall not make assumptions about the validity of any out-parameters.
- When `kHAPError_None` is returned, all applicable out parameters of the function must contain valid values.

Specification References

- Always reference the spec if possible, and always use the exact same format to simplify "Find in Files" searches.
 - First line: `@see` or See followed by exact PDF Title as shown in title bar when opening it.
 - Second line: Section + Section number + exact Section Title.
 - Tables may be referenced as well, in that case use Table instead of Section.
 - Section references are preferred as tables may be harder to find.
- For code comments, quote exactly what part of the spec is referenced.

- Always use the most recent specification version where the feature is documented. If a feature becomes obsolete, use the Doxygen `@obsolete` tag to specify the first revision it got removed. Avoid actually deleting enum values etc, use `@obsolete` instead.

Example for documentation block

```
/**  
 * ...  
 *  
 * @see HomeKit Accessory Protocol Specification R15  
 *     Section 1.2.3.4 Some Sample Section  
 */
```

Example for code comment

```
// ...  
// See HomeKit Accessory Protocol Specification R15  
// Section 1.2.3.4 Some Sample Section
```

Definitions

- Definition name: `kHAPCategory_PascalCase`
- Always explicitly specify the type of the definition, e.g., `uint8_t`, `size_t`.
- Consider using enumerations instead, where possible.

```
/**  
 * Brief description  
 */  
# define kHAPCategory_PascalCase ((size_t) 0x42)  
<p class="mume-header" id="define-khapcategory_pascalcase-size_t-0x42"></p>
```

Enumerations

- Enumerations are never used to store out-of-range values. Instead, the underlying type is used, e.g., `uint8_t`. Before casting to an enum, perform proper input validation. When receiving an enum, no thorough input validation is necessary as it is already done.
- Type name: `HAPPascalCase`
- Case name: `kHAPTypeName_PascalCase` Always prefix with lower case `k`.
- If there is a default, uninitialized, 0 case, list it first, but do not set explicitly to 0.
- If there is no default case, set first value to 1 to avoid accidents with `memset 0`. Do not set values to other cases.
- For flags, set the initial case to `1 << 0`, the next to `1 << 1` etc.
- Align documentation on column 37 if possible, otherwise use a full separate Doxygen style comment block. If there are some long and some short cases, consider using the full comment block for all cases. A single-line Doxygen style comment block may be used to condense information. If the enum is defined in an indented scope, columns advance by 4 per indentation.
- In interfaces, always use `HAP_ENUM_BEGIN` / `HAP_ENUM_END` for portability across different compiler settings.

```

/**
 * Enum type documentation.
 */
HAP_ENUM_BEGIN(uint8_t, HAPEnumType) {
    kHAPEnumType_PascalCase1,      /**< Case 1 documentation. */
    kHAPEnumType_PascalCase2,      /**< Case 2 documentation. */

    /**
     * Case 3 documentation.
     *
     * - Remark 1.
     *
     * - Remark 2.
     */
    kHAPEnumType_ReallyReallyLongCase,
}

/** Case 4 documentation. */
kHAPEnumType_ReallyReallyLongCase2
} HAP_ENUM_END(uint8_t, HAPEnumType);

```

Functions

- Function names: *HAPPascalCase*

Use concise names and avoid abbreviations. Length does not matter here. If targeting an object, first mention the object, e.g., `HAPAccessoryServerStartBridge` targets a `HAPAccessoryServer` object. In that case, also make sure that the target object is the very first parameter.

- Parameter names: *camelCase*

Maximum parameter length name is 20. Otherwise it will break the table in the documentation.

Use `HAP_UNUSED`, if the argument is not used in the function body. Do not cast to `(void)` for compiler warnings.

Pointers are assumed to always be non-NULL. Use `_Nullable` on pointer arguments that may be NULL.

- Parameter types use fixed size C99 integers, e.g., `uint8_t`, `int16_t`, `size_t`. Avoid `int`, `short`, `unsigned char`. For booleans, use `bool`. Yes, it's just a typedef to `int` in C, but it improves readability.
- Return type must be `HAPError` if the function is expected to fail. If the function returns a value (non-void), it must be marked with `HAP_RESULT_USE_CHECK`. If the function cannot fail and does not provide any outputs, return `void`. If unsure whether a function may return an error, start with `void` return first. Errors that arise from API misuse (invalid arguments, invalid state) crash / assert instead of returning error.
- Every parameter and return value is documented.
 - If a parameter is an out-parameter, `[out]` is attached after the `@param`.
 - If a parameter is an in/out parameter, `[in, out]` is attached after the `@param`.
 - `true` / `false` return values are written in lowercase in documentation.
 - `HAPError` return values are sorted in documentation in the same way as listed in the `HAPError` enum.
- GCC attributes one per line, sorted alphabetically. Apply GCC attributes to both declaration and definition of the function.
- Function bodies are streamlined:
 - Precondition checks: One precondition per line, checked in the order they appear in the function signature. Casts from Refs to internal types should be made as close as possible to where their preconditions are checked. Early returns may also be included as part of the precondition checks block if it makes sense.
 - `HAPError err;`
 - Function body.
- Use early returns if possible.

If returning because of an error, use a `HAPLog` statement to print the reason of the error before throwing. When just re-throwing an error, do not log. The error is already logged when it was thrown. If a post-condition must be checked even in error cases, wrap the function body in a second function that uses early returns, and check the post-condition in the wrapping function.

```

/***
 * Brief documentation about the function.
 *
 * - Remark 1 about special considerations.
 *
 * - Remark 2 about special considerations.
 *
 * @param arg1 Argument 1 documentation.
 * @param arg2 Argument 2 documentation.
 * @param[out] bytes Buffer that will be filled with data.
 * @param maxBytes Maximum number of bytes that may be filled into the buffer.
 * @param[in,out] numBytes Effective number of bytes written to the buffer.
 *
 * @return kHAPError_None If successful.
 * @return kHAPError_OutOfResources If the supplied buffer was not large enough.
 */
HAP_RESULT_USE_CHECK
static HAPError HAPDoSomething(
    size_t arg1,
    bool arg2 HAP_UNUSED,
    void *bytes, // _Nullable if bytes are _Nullable
    size_t maxBytes,
    size_t *numBytes) // _Nullable if bytes are _Nullable
{
    HAPPrecondition(arg1 > 42);
    HAPPrecondition(bytes);

    HAPError err;

    // ...

    return kHAPError_None;
}

```

Header Files

- Name: `HAPPascalCase.h`. The file name does not have to be a prefix of all functions.
- If several items form a certain subgroup of a header, consider using a separate header file.
Name: `HAPBaseHeader+CategoryName.h`
- Include header files from `HAP.h`. Include category headers from their corresponding base header.
- Header guards: `HAP_UNDERSCORED_FILE_NAME_H`. Do not forget to update the header guard after file renames. Do not forget to update the log category / subsystem in source files that implement the header after file renames.
- Always include `extern "C"` declaration, even in internal headers. Headers may move around over time and become visible to C++. Also include the declaration when it is not necessary, for consistency. It doesn't hurt.
- Header files are streamlined:
 - Copyright notice. Make sure it is EXACTLY the same in all files, so that automated re-copyrighting scripts work.
 - Empty line.
 - Header guard.
 - Empty line.
 - `extern "C"` declaration.
 - `#include <system_header.h>`, sorted alphabetically. If some are in a subdirectory, place them in the end.
System headers must only be included in `HAPBase.h` and in platform-specific code!
 - Empty line, if system headers were included.
 - `#include "platform_header.h"`, sorted alphabetically. This is for headers coming from external dependencies.
 - Empty line, if platform headers were included.
 - `#include "HAP.h" / #include "HAPPlatform.h"`, and categories of own header files, sorted alphabetically.
 - Empty line.
 - Enter `assume_nonnull` block.
 - Header file contents.
 - Exit `assume_nonnull` block.
 - Empty line.
 - If necessary, further `#include` statements, followed by empty line.

- Complete extern "C" declaration.
- Empty line.
- Complete header guard.
- Empty line.

```
# ifndef HAP_HEADER_FILE_NAME_H
<p class="mume-header" id="ifndef-hap_header_file_name_h"></p>

# define HAP_HEADER_FILE_NAME_H
<p class="mume-header" id="define-hap_header_file_name_h"></p>

# ifdef __cplusplus
<p class="mume-header" id="ifdef-__cplusplus"></p>

extern "C" {
# endif
<p class="mume-header" id="endif-6"></p>

# include <system_header.h>
<p class="mume-header" id="include-system_headerh"></p>

# include "platform_header.h"
<p class="mume-header" id="include-platform_headerh"></p>

# include "HAPPlatform.h"
<p class="mume-header" id="include-happlatformh"></p>

# if __has_feature(nullability)
<p class="mume-header" id="if-__has_featurenullability"></p>

# pragma clang assume_nonnull begin
<p class="mume-header" id="pragma-clang-assume_nonnull-begin"></p>

# endif
<p class="mume-header" id="endif-7"></p>

// Header file contents.

# if __has_feature(nullability)
<p class="mume-header" id="if-__has_featurenullability-1"></p>

# pragma clang assume_nonnull end
<p class="mume-header" id="pragma-clang-assume_nonnull-end"></p>

# endif
<p class="mume-header" id="endif-8"></p>

# ifdef __cplusplus
<p class="mume-header" id="ifdef-__cplusplus-1"></p>

}
# endif
<p class="mume-header" id="endif-9"></p>

# endif
<p class="mume-header" id="endif-10"></p>
```

Switch Statements

- When switching over a closed enum, do not add a default case. Try to return from every case, and place a fatal error

after the switch. This ensures compiler warnings when new enum cases are added but are not explicitly handled.

- Use a separate scope for every case. (Empty cases are allowed to be condensed together, e.g., case 'c' / 'd' below).
- After the closing bracket of each case's scope, put either a `break;` statement or a `// Fallthrough` comment.
- When switching on enumerations, add a default case like `default: HAPFatalError();` or `default: false` as the final line to cover unexpected values.

```
switch (foo) {  
    case 'a': {  
        // Handle case.  
        break;  
    }  
    case 'b': {  
        // Handle case.  
        break;  
    }  
    case 'c':  
    case 'd': {  
        // Handle cases.  
    } // Fallthrough.  
    case 'e': {  
        // Handle case.  
        break;  
    }  
}
```

- Exceptions are simple switch statements with early returns where putting an extra break statement after the case's scope does not help with readability. If there are a lot of cases and the switch is just a conversion table from one constant to another it can also be considered to remove the brackets to avoid spanning the switch across pages.

```
static const char *GetCurrentHeatingCoolingStateDescription(  
    HAPCharacteristicValue_CurrentHeatingCoolingState state)  
{  
    switch (state) {  
        case kHAPCharacteristicValue_CurrentHeatingCoolingState_Off:  
            return "Off.";  
        case kHAPCharacteristicValue_CurrentHeatingCoolingState_Heat:  
            return "Heat. The Heater is currently on.";  
        case kHAPCharacteristicValue_CurrentHeatingCoolingState_Cool:  
            return "Cool. Cooler is currently on.";  
        default: HAPFatalError();  
    }  
}
```

Goto Statements

- In general, goto statements should be avoided. Especially, goto statements must not be used to jump backwards. However, in the following scenarios, goto statements may be used.
- If a postcondition needs to be checked or final cleanup needs to be performed before returning from a function, instead of using an early return, a goto statement may be used to handle that finalization.
- Goto statements may be used to break out of nested loops.
- Goto statements may also be used to break from a case statement of a switch on an enumeration. This allows placing a `HAPFatalError()` immediately after the switch without providing a default case. Not providing a default case allows the compiler to emit warnings if an enumeration value is not handled. Placing a `HAPFatalError()` and goto label immediately after the switch allows catching unexpected value.
- Goto statements may be used when multiple cases of a switch prepare some variables but otherwise share their implementation. In this case the goto label shall be placed immediately after the cases sharing the code, before the definition of the next case label.
- Goto labels should use camelCase.

Magic numbers

- Number literals are avoided in favor of named constants, except where a literal is more transparent, e.g.:

- obvious offsets like +/- 1
- offsets or shift distances and masks for bit manipulation like this: `mantissa = (x >> 9) & 0xFFFF;`
- POSIX success (0) and most important error (-1) numbers
- Test code is allowed to contain magic numbers.

Dynamic memory allocation

- There are only a few places in the PAL where we do dynamic memory allocation, none in the HAP Library.
- Use the macro `HAPPlatformFreeSafe` to deallocate memory in cases where memory is allocated dynamically with libc.
- Use similar macros for third-party library deallocation.

Use of parameters

- All public functions are documented with Doxygen comments.
- All const pointers and non-pointer values are inputs (caller provides data).
- All non-const pointers are outputs or inouts.
- In Doxygen, parameters where the caller must supply input data are not annotated.
- In Doxygen, parameters where the caller should use output data are annotated with [out].
- In Doxygen, parameters where the caller must supply input data and should use output data are annotated with [in,out], except for "this" pointers which are the first parameters of functions that work with object-like data types.

Use of void parameters

- We normally use typed parameters.
- If byte buffers are returned, void is used instead, to not force clients to use type casts. This is the same style as Apple's mDNSResponder.

Opaque structures

- Internal structures that are exposed publicly should be hidden by following the opaque pattern.
- Public header: Use the `HAP_OPAQUE` define with the first arguments defining the size in multiples of 8. The associated data is aligned on 8 bytes. Name ends in `Ref`.

```
typedef HAP_OPAQUE(24) HAPFooRef;
```

- Private header: Specify real structure without "Ref" in name and static assert that it fits into a Ref.

```
typedef struct { ... } HAPFoo;
```

```
HAP_STATIC_ASSERT(sizeof (HAPFooRef) >= sizeof (HAPFoo), HAPFoo);
```

- Functions: Always pass around refs. If internal fields need to be accessed, create second variable with `_` postfix.

```
HAPFooRef *foo_;
HAPFoo *foo = (HAPFoo *) foo_;
```

Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies within all project spaces, and it also applies when an individual is representing the project or its community in public spaces.

Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the open source team at opensource-conduct@group.apple.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the [Contributor Covenant](#), version 1.4, available at <https://www.contributor-covenant.org/version/1/4/code-of-conduct.html>

The following organizations and individuals have contributed to the HomeKit ADK:

- Oberon microsystems AG
- Apple Inc.

By submitting a pull request, you represent that you have the right to license your contribution to Apple and the community, and agree by submitting the patch that your contributions are licensed under the [Apache 2.0 license](#).

Before submitting the pull request, please make sure you have tested your changes and that they follow the project [guidelines for contributing code](#).

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition,

"control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally

submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution." "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or

documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions.

Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability

incurred by, or claims asserted against, such Contributor by reason
of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

HomeKit Accessory Development Kit (ADK)

The HomeKit ADK is used by silicon vendors and accessory manufacturers to build HomeKit compatible devices.

The HomeKit ADK implements key components of the HomeKit Accessory Protocol (HAP), which embodies the core principles Apple brings to smart home technology: security, privacy, and reliability.

The HomeKit Open Source ADK is an open-source version of the HomeKit Accessory Development Kit. It can be used by any developer to prototype non-commercial smart home accessories. For commercial accessories, accessory developers must continue to use the commercial version of the HomeKit ADK available through the MFi Program.

Go to the [Apple Developer Site](#) if you like to learn more about developing HomeKit-enabled accessories and apps.

Documentation

- Please go through [Developing with ADK](#) before starting development with HomeKit ADK
- [Platform Abstraction Layer](#)

Darwin PAL

Compile

`make all`

Run

Linux PAL

Compile

```
make TARGET=Linux apps
```

Raspberry Pi

Prerequisites

Run this to create the SD card image (Linux+patches) and a Docker container (your build environment).

Make sure that Docker is running before running this setup or it will fail and you will have to start over!

This will take about an hour and will stop for input many times. It will also require you to swap the card out twice.

```
# Run the Docker app. It is required for docker import.  
<p class="mume-header " id="run-the-docker-app-it-is-required-for-docker-import"></p>  
./Tools/raspi_sdcard_setup.sh
```

Compile

```
make TARGET=Raspi all
```

If docker doesn't find "dev-test/raspriadk-base", run the sdcard setup and make sure to do the docker import at the end of the script.

Install

After building, run this to install the build products to your RaspPi. (-n is for the hostname of your RaspPi and -p is the SSH password; both of these were chosen during the initial raspi_sdcard_setup.sh install).

```
./Tools/install.sh \  
-d raspi \  
-a Output/Raspi-armv6k-unknown-linux-gnueabihf/Debug/IP/Applications/Lightbulb.OpenSSL \  
-n raspberrypi \  
-p pi
```

Make options

Command	Description	Default
make <target>	<ul style="list-style-type: none">apps - Build all appstest - Build unit testsall - Build apps and unit tests	all
make APPS=<application>	Space delimited names of the apps to compile. Example: <code>make APPS="Lightbulb Lock"</code>	All applications
make BUILD_TYPE=<build_type>	Build type: <ul style="list-style-type: none">DebugTestRelease	Debug
	<ul style="list-style-type: none">0 - No logs are displayed1 - Error and Fault-level logs are displayed	<ul style="list-style-type: none">3 - For debug build1 - For test build

make LOG_LEVEL=<level>	<ul style="list-style-type: none"> • 2 - Error, Fault-level and Info logs are displayed • 3 - Error, Fault-level, Info and Debug logs are displayed 	• 0 - For release build
make PROTOCOLS=<protocol>	<p>Space delimited protocols supported by the applications:</p> <ul style="list-style-type: none"> • BLE • IP <p>Example: make PROTOCOLS="IP BLE"</p>	All protocols
make TARGET=<platform>	<p>Build for a given target platform:</p> <ul style="list-style-type: none"> • Darwin • Linux • Raspi 	Build for the host Platform
make USE_HW_AUTH=<enable>	<p>Build with hardware authentication enabled:</p> <ul style="list-style-type: none"> • 0 - Disable • 1 - Enable 	Disabled
make USE_NFC=<enable>	<p>Build with NFC enabled:</p> <ul style="list-style-type: none"> • 0 - Disable • 1 - Enable 	Disabled

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can

be assigned to you.

2. Follow the normal process of [forking](#)

the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.

3. Any significant changes should almost always be accompanied by tests. The

project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.

4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.

5. Do your best to have [well-formed commit messages](#)

for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.

6. Finally, push the commits to your fork and submit a [pull request](#).

INA3221 I2C Driver

A Mongoose library for Texas Instruments' 3-channel current sensor.

Implementation details

The `INA3221` is a simple I2C device that measures current through shunt resistors on three separate channels. It is able to measure the voltage on each channel (max 26V, in 8mV increments) and the voltage drop over the shunt resistor in 40uV increments with a range of 163.8mV, and is quite precise.

Using a 0.1Ohm shunt resistor, the maximum current that can be measured is 1.638A. Each channel can be enabled/disabled and polled individually.

API Description

Create an `INA3221` object using `mgos_ina3221_create()`, set the shunt resistor values for each channel using `mgos_ina3221_set_shunt_resistance()`, then call `mgos_ina3221_get_bus_voltage()` to get the bus voltage, call `mgos_ina3221_get_shunt_voltage()` to get the voltage drop over the shunt resistor, and `mgos_ina3221_get_current()` to get the current flowing through the channel.

Valid channels are 1, 2 or 3 (ie. they are 1-based).

Example application

```

#include "mgos.h"
<p class="mume-header" id="include-mgosh-26"></p>

#include "mgos_config.h"
<p class="mume-header" id="include-mgos_configh-5"></p>

#include "mgos_ina3221.h"
<p class="mume-header" id="include-mgos_ina3221h"></p>

static void ina3221_timer_cb(void *user_data) {
    struct mgos_ina3221 *sensor = (struct mgos_ina3221 *)user_data;

    if (!sensor) return;

    for (uint8_t i=1; i<4; i++) {
        float bus, shunt, current, res;
        mgos_ina3221_get_bus_voltage(sensor, i, &bus);
        mgos_ina3221_get_shunt_resistance(sensor, i, &res);
        mgos_ina3221_get_shunt_voltage(sensor, i, &shunt);
        mgos_ina3221_get_current(sensor, i, &current);
        LOG(LL_INFO, ("Chan[%u]: Vbus=% .3f V Vshunt=% .0f uV Rshunt=% .3f Ohm Ishunt=% .1f mA",
                     i, bus, shunt*1e6, res, current*1e3));
    }
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_ina3221 *sensor;

    sensor = mgos_ina3221_create(mgos_i2c_get_global(), mgos_sys_config_get_ina3221_i2caddr());
    if (!sensor) {
        LOG(LL_ERROR, ("Could not create INA3221 sensor"));
        return false;
    }

    mgos_set_timer(1000, true, ina3221_timer_cb, sensor);

    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Debug over Bluetooth GATT Service

Overview

This library provides various debug functions over Generic Attribute Service (GATT) Bluetooth Low-Energy (BLE) service.

The service is designed to be usable with any generic BLE mobile app that supports GATT, e.g. BLE Scanner ([Android](#), [iOS](#)).

Note: Default BT configuration is permissive. See <https://github.com/mongoose-os-libs/bt-common#security> for a better idea.

Attribute description

The service UUID is `5f6d4f53-5f44-4247-5f53-56435f49445f`, which is a representation of a 16-byte string `_mOS_DBG_SVC_ID_`.

At present, only one characteristic is defined:

- 306d4f53-5f44-4247-5f6c-6f675f5f5f30 (0mOS_DBG_log__0) - a read/notify attribute that returns last debug log record when read. It also sends notifications with log messages as they are printed.
 - Note: Reading large messages is supported, but for notifications to be useful you will most likely want to set higher MTU.

File-logger

File-logger is a library which implements log file rotation: it maintains max X log files of max size Y, so that you always have latest logs from the device persisted on the filesystem. By default there are max 10 files, prefixed with `log_`, each of max size 4000 bytes.

See [mos.yml](#) for the possible options. At least you'd have to enable this lib in your app's `mos.yml`, like this:

Note that for the message to get to file logger it needs to be allowed by `debug.level` and `debug.event_level`. File logger then examines `file_logger.level` and `file_logger.include` to make final determination.

`file_logger.include` is a list of substrings to match, so a value of `mg_rpc.c,Tick,Tock` will match lines containing any of "Tick", "Tock" or "mg_rpc.c".

```
libs:  
  - origin: https://github.com/mongoose-os-libs/file-logger  
  
config_schema:  
  - ["file_logger.enable", true]
```

Fetching logs

See [this script](#). libbmpread

libbmpread is a tiny, fast bitmap (.bmp) image file loader, written from scratch in portable C (see below), with no dependencies. Its default behavior is compatible with OpenGL texture functions, making it ideal for use in simple games. It handles any valid bit depth (1, 4, 8, 16, 24, or 32), and can even load the alpha channel from 16- and 32-bit bitmaps. There is no support for RLE or other compression schemes yet.

<https://github.com/chazomaticus/libbmpread>

Documentation

To use, simply copy `bmpread.c` and `bmpread.h` into your project and add them to the build.

The API exposed by `bmpread.h` is described below. The same description can be found in the header itself.

bmpread()

Loads the specified bitmap file from disk and fills out a `bmpread_t` struct

with data about it.

```
int bmpread(const char * bmp_file,
            unsigned int flags,
            bmpread_t * p_bmp_out);
```

- `bmp_file`: The filename of the bitmap file to load.
- `flags`: Any `BMPREAD_*` flags, combined with bitwise OR. Specify 0 (or `BMPREAD_ALPHA` if you want an alpha channel) for standard, OpenGL compliant behavior.
- `p_bmp_out`: Pointer to a `bmpread_t` struct to fill with information. Its contents on input are ignored. Must be freed with `bmpread_free()` when no longer needed.

Returns 0 if there's an error (file doesn't exist or is invalid, i/o error, etc.), or nonzero if the file loaded ok.

The file must be a Windows 3 (not NT) or higher format bitmap file with any valid bit depth (1, 4, 8, 16, 24, or 32), and must not be compressed (no RLE).

Default behavior is for `bmpread()` to return `data` in a format directly usable by OpenGL texture functions, e.g. `glTexImage2D`, format `GL_RGB` (or `GL_RGBA` if `BMPREAD_ALPHA` is in `flags`), type `GL_UNSIGNED_BYTE`. This implies a few oddities:

- Lines are ordered bottom-first. To return data starting with the top line like you might otherwise expect, pass `BMPREAD_TOP_DOWN` in `flags`.
- Lines are padded to span a multiple of four bytes. To return data with no padding, pass `BMPREAD_BYTE_ALIGN` in `flags`.
- Images with a width or height that isn't a power of 2 will fail to load. To allow loading images of any size, pass `BMPREAD_ANY_SIZE` in `flags`.

Note that passing any of these flags may cause the output to be unusable as an OpenGL texture, which may or may not matter to you.

Most bitmap files can't include an alpha channel, so the default behavior is to ignore any alpha values present in the file. Pass `BMPREAD_ALPHA` in `flags` to capture alpha values from the file; in case of an absent alpha channel, alpha values are output as 255 (this can be changed by redefining `BMPREAD_DEFAULT_ALPHA` in `bmpread.c`). This allows fully loading 16- and 32-bit bitmaps, which *can* include an alpha channel.

bmpread_free()

Frees memory allocated during `bmpread()`. Call `bmpread_free()` when you are done using the `bmpread_t` struct (e.g. after you have passed the data on to OpenGL).

```
void bmpread_free(bmpread_t * p_bmp);
```

- `p_bmp`: The pointer you previously passed to `bmpread()`.

bmpread_t

The struct filled by `bmpread()`. Holds information about the image's pixels.

```

typedef struct bmpread_t
{
    int width;
    int height;

    unsigned int flags;

    unsigned char * data;

} bmpread_t;

```

- `width`: Width in pixels.
- `height`: Height in pixels.
- `flags`: `BMPREAD_*` flags, combined with bitwise OR, that affect the format of `data`. These are set to the flags passed to `bmpread()`.
- `data`: A buffer holding the pixel data of the image.

By default, each pixel spans three bytes: the red, green, and blue color components in that order. However, with `BMPREAD_ALPHA` set in `flags`, each pixel spans four bytes: the red, green, blue, and alpha components in that order.

Pixels are ordered left to right sequentially. By default, the bottom line comes first, proceeding upward. However, with `BMPREAD_TOP_DOWN` set in `flags`, the top line comes first, proceeding downward instead.

Lines by default must span a multiple of four bytes. If the image width and pixel span don't yield a multiple of four (a non-issue for `BMPREAD_ALPHA` with four bytes per pixel), the end of each line is padded with up to three unused bytes to meet the requirement. For example, each line of an image three pixels wide, loaded without `BMPREAD_ALPHA`, will span 12 bytes (3 pixels * 3 (RGB) channels per pixel = 9, padded with 3 bytes up to the next multiple of 4). However, this behavior is disabled with `BMPREAD_BYTE_ALIGN` set in `flags`, in which case all lines span exactly `width * pixel_span` bytes.

Flags

Flags for `bmpread()` and `bmpread_t`. Combine with bitwise OR.

- `BMPREAD_TOP_DOWN`: Output data as top line first (default is bottom line first).

```
#define BMPREAD_TOP_DOWN 1u
```

- `BMPREAD_BYTE_ALIGN`: Don't pad lines to span a multiple of four bytes (default does pad).

```
#define BMPREAD_BYTE_ALIGN 2u
```

- `BMPREAD_ANY_SIZE`: Allow loading of any size bitmap (default is bitmaps must be $2^n \times 2^m$).

```
#define BMPREAD_ANY_SIZE 4u
```

- `BMPREAD_ALPHA`: Load and output an alpha channel (default is just color channels).

```
#define BMPREAD_ALPHA 8u
```

Example

Here's a code snippet showing how libbmpread might be used to create an OpenGL texture from a bitmap file on disk:

```
# include <stdio.h>
<p class="mume-header" id="include-stdioh"></p>

# include <GL/gl.h>
<p class="mume-header" id="include-glgih"></p>

# include "bmpread.h"
<p class="mume-header" id="include-bmpreadh"></p>

/*
 * Load the specified bitmap file from disk and copy it into an OpenGL texture.
 * Return the GLuint representing the texture.
 */
GLuint LoadTexture(const char * bitmap_file)
{
    GLuint texture = 0;
    bmpread_t bitmap;

    if(!bmpread(bitmap_file, 0, &bitmap))
    {
        fprintf(stderr, "%s: error loading bitmap file\n", bitmap_file);
        exit(1);
    }

    /* At this point, bitmap.width and .height hold the pixel dimensions of the
     * file, and bitmap.data holds the raw pixel data in RGB triplets.
     */

    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    glTexImage2D(GL_TEXTURE_2D, 0, 3, bitmap.width, bitmap.height, 0,
                 GL_RGB, GL_UNSIGNED_BYTE, bitmap.data);

    bmpread_free(&bitmap);

    return texture;
}

void SomeInitFunction(void)
{
    GLuint tex1 = LoadTexture("texture1.bmp");
    // ...
}
```

See also the `example` directory for a full, compiling example that displays a bitmap named on the command line on an OpenGL quad using GLUT. To build and run:

```
cd example
make
./bmpread-example <bmpfile>
```

Some example bitmap files are included there as well.

Portability/Security Notes

libbmpread is written to be maximally ANSI C (C89/C90) portable, minimizing undefined and implementation-defined behavior. It's also written to be able to be compiled as C++, so it can be used in C++ projects with a minimum of fuss.

It should work in most environments--I believe there are only two assumptions the code makes that aren't guaranteed by the C standard: 1) `CHAR_BIT == 8` (note: if this is violated, compilation will fail), and 2) two's complement integer storage; deviations from these assumptions are extremely rare in the wild. I've also made use of `<stdint.h>` and `ptrdiff_t`, which I believe are technically C99 features, but are common in practice even for non-compliant compilers.

I've taken every precaution to prevent common bugs that can have security impact, such as integer overflows that might lead to buffer overruns. I believe it's impossible to cause libbpread to do anything besides properly load a file or fail with error, even on maliciously crafted files. I haven't done a thorough audit, nor am I an expert at writing hardened "C/C++" code, so take my belief with a grain of salt.

Tests

To run the test suite:

```
cd test  
make
```

The last line will read `All tests passed!` if everything was ok. Be patient: I test an unoptimized function over its entire 32-bit numeric range, so it can take a minute or two to finish.

There are some basic unit tests, but more work is needed to test the behavior of the public API. The tests are compiled as both portable C and portable C++ code to cover compilation in each, and each resulting test binary is run to check its runtime behavior.

Authors

- [Charles Lindsay](#) (maintainer)
- [xFrednet](#) (added support for 16- and 32-bit files)

Enjoy!

Wrapper lib for Mongoose-OS to load BMP images

This is a Mongoose OS library to use "[libbpread](#)" by Charles Lindsay "chazomaticus" seamlessly in Mongoose OS applications.

The third-party software is integrated without modification under the directory `third_party`. The documentation for the underlying lib can also be found in the `third_party` directory. The external library is 100% original and not changed (will be included as submodule in a future version), so it's easier to integrate updates later, if they show up.

Be aware: this library might be subject of change, so use it at your own risk!

Dependencies

This library has a dependency:

- [libbpread](#)

Documentation

These functions are available via C calls:

```
// create the needed structure to hold the data:  
bmpread_t* mgos_bmp_loader_create(void)  
  
// load an BMP image  
bool mgos_bmp_loader_load(bmpread_t* p_bmp, const char* bmp_file, unsigned int flags)  
  
// free the allocated space  
void mgos_bmp_loader_free(bmpread_t* p_bmp)  
  
// get the plain data pointer of the image  
uint8_t* mgos_bmp_loader_get_data(bmpread_t* p_bmp)  
  
// get the width and height of the image (wrappers for FFI)  
uint32_t mgos_bmp_loader_get_width(bmpread_t* p_bmp)  
uint32_t mgos_bmp_loader_get_height(bmpread_t* p_bmp)  
  
// rotate the image in 90° steps  
bool mgos_bmp_loader_rotate(bmpread_t* p_bmp, uint16_t angle)
```

A MJS API is included as well:

Include the API with

```
load('api_bmp.js');
```

in your own MJS file like `init.js`.

These functions are available as MJS API calls:

```
// the object containing the API
let BMP_LOADER = { ... };

// create a new object with the needed structure
create: function () { ... }

// load a BMP file
load: function (file, flags) { ... }

// free the underlying structure and the bitmap data
free: function () { ... }

// get a pointer to the raw bitmap data
getData: function () { ... }

// get a certain pixel of the bitmap as pointer to the bytes
getPixel: function (x, y) { ... }

// get the width of the image
getWidth: function () { ... }

// get the height of the image
getHeight: function () {...}
```# Inventek es-WiFi module driver for Mongoose OS
```

Driver for the [Inventek eS-WiFi ISM43xxx module series](<https://www.inventeksys.com/es-wifi-support/>) (ISM4319-M3-xxx, ISM

Only SPI interface is supported for now.

## ## Configuration

In addition to the common wifi STA and AP options, this library adds a number of its own settings.

You need to configure at least CS and DRDY pins, others are optional.

See [manifest](mos.yml) for definitions and examples.

```
SNTP library for Mongoose OS
```

When included to a Mongoose OS application, this library fetches the current time from the SNTP server (by default, `time.google.com` is used) every time the Internet connection is established, and adjusts the device time.

See `mos.yml` for the SNTP configuration available.

```
Board definitions library
```

This library contains configuration tweaks for different boards supported by Mongoose OS.

```
Python CryptoAuthLib Module Testing
```

## ## Introduction

These tests are designed to only test the python interface to the library and are not designed to test the library itself which is covered by the main cryptoauthlib tests

## ### Running

The best way to run the test suite is to use [tox](<https://tox.readthedocs.io/en/latest/>) which can be easily installed with pip:

```
$ pip install tox
```

From the python folder:

```
:~/cryptoauthlib/python $ tox
```

It is possible to directly run tests but requires more setup

### 1) Install pytest

```
$ pip install pytest
```

### 2) Modify the PYTHONPATH environment variable

Windows:

```
cryptoauthlib/python> set PYTHONPATH=<path_to>/cryptoauthlib/python
```

Linux:

```
export PYTHONPATH={PYTHONPATH}:<path_to>/cryptoauthlib/python
```

### 3) Run the tests

```
$ pytest -vv
```

#### ### Test options

There are additional options that can be invoked with the tests that define what tests will be run

1) --with-lib will attempt to run tests against the compiled c library.

These tests are good for detecting possible platform incompatibilities between the C compiler and the expectations of python

2) --with-device will attempt to invoke some tests with a real attached device

These tests are restricted to only the minimum required to verify the python to library connectivity and are only meant to detect situations that can not be determined from the library tests alone.

```
Python CryptoAuthLib module
```

#### ## Introduction

This module provides a thin python ctypes layer to evaluate the cryptoauthlib interface to Microchip CryptoAuthentication devices.

#### ### Code Examples

Code examples for python are available on github as part of

[CryptoAuthTools](<https://github.com/MicrochipTech/cryptoauthtools>)

under the [python/examples](<https://github.com/MicrochipTech/cryptoauthtools/tree/master/python/examples>) directory

#### ## Installation

### CryptoAuthLib python module can be installed through Python's pip tool:

```
pip install cryptoauthlib
```

### To upgrade your installation when new releases are made:

```
pip install -U cryptoauthlib
```

### If you ever need to remove your installation:

```
pip uninstall cryptoauthlib
```

## What does python CryptoAuthLib package do?

CryptoAuthLib module gives access to most functions available as part of standard cryptoauthlib (which is written in 'C'). These python functions for the most part are very similar to 'C' functions. The module in short acts as a wrapper over the 'C' cryptoauth library functions.

Microchip cryptoauthlib product page:

[Link]( <http://www.microchip.com/SWLibraryWeb/product.aspx?product=CryptoAuthLib>)

## Supported hardware

- [AT88CK101](<http://www.microchip.com/DevelopmentTools/ProductDetails/AT88CK101SK-MAH-XPRO>)

- [CryptoAuthentication SOIC XPRO Starter Kit (DM320109)](<https://www.microchip.com/developmenttools/ProductDetails/DM320109>)

## Supported devices

The family of devices supported currently are:

- [ATSHA204A](<http://www.microchip.com/ATSHA204A>)
- [ATECC108A](<http://www.microchip.com/ATECC108A>)
- [ATECC508A](<http://www.microchip.com/ATECC508A>)
- [ATECC608A](<http://www.microchip.com/ATECC608A>)

## Using cryptoauthlib python module

The following is a 'C' code made using cryptoauthlib 'C' library.

```
```C
#include "cryptoauthlib.h"
<p class="mume-header" id="include-cryptoauthlib"></p>

void main()
{
    ATCA_STATUS status;
    uint8_t revision[4];
    uint8_t randomnum[32];

    status = atcab_init(cfg_ateccx08a_kitcdc_default);
    if (status != ATCA_SUCCESS)
    {
        printf("Error");
        exit();
    }

    status = atcab_info(revision);
    if (status != ATCA_SUCCESS)
    {
        printf("Error");
        exit();
    }

    status = atcab_random(randomnum);
    if (status != ATCA_SUCCESS)
    {
        printf("Error");
        exit();
    }
}
```

The same code in python would be:

```

from cryptoauthlib import *

ATCA_SUCCESS = 0x00
revision = bytearray(4)
randomnum = bytearray(32)

# Locate and load the compiled library
<p class="mume-header " id="locate-and-load-the-compiled-library"></p>

load_cryptoauthlib()

assert ATCA_SUCCESS == atcab_init(cfg_ateccx08a_kithid_default())

assert ATCA_SUCCESS == atcab_info(revision)
print(''.join(['%02X ' % x for x in revision]))

assert ATCA_SUCCESS == atcab_random(randomnum)
print(''.join(['%02X ' % x for x in randomnum]))

```

In the above python code, "import cryptoauthlib" imports the python module. `load_cryptoauthlib()` function loads the compiled library. The `load_cryptoauthlib()` is a function that you will not see in the 'C' library, this is a python specific utility function and is required for python scripts to locate and load the compiled library.

In Summary

Step I: Import the module

```
from cryptoauthlib import *
```

Step II: Initialize the module

```

load_cryptoauthlib()

assert ATCA_SUCCESS == atcab_init(cfg_ateccx08a_kithid_default())

```

Step III: Use Cryptoauthlib APIs

Call library APIs of your choice

Code portability

Microchip's CryptoAuthentication products can now be evaluated with the power and flexibility of python. Once the evaluation stage is done the python code can be ported to 'C' code.

As seen above the python API maintains a 1 to 1 equivalence to the 'C' API in order to ease the transition between the two.

Cryptoauthlib module API documentation

help() command

All of the python function's documentation can be viewed through python's built in `help()` function.

For example, to get the documentation of `atcab_info()` function:

```
>>> help(cryptoauthlib.atcab_info)
```

```
Help on function atcab_info in module cryptoauthlib.atcab:
```

```
atcab_info(revision)
```

```
Used to get the device revision number. (DevRev)
```

```
Args:
```

```
    revision      4-byte bytarray receiving the revision number  
                  from the device. (Expect bytarray)
```

```
Returns:
```

```
    Status code
```

dir() command

The dir command without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object. For example
dir(cryptoauthlib) will return all the methods available in the cryptoauthlib module.

Code Examples

Code examples for python are available on github as part of

[CryptoAuthTools](#) under the

python/examples directory

Tests

Module tests can be located in the [python/tests](#)

of the main cryptoauthlib repository. The [README.md](#)

has details for how to run the tests. The module tests are not comprehensive for the entire functionality of cryptoauthlib but rather are meant to test the python module code only against the library to ensure the interfaces are correct and ctypes structures match the platform.

basic directory - Purpose

The purpose of this directory is to contain the files implementing the APIs for a basic interface to the core CryptoAuthLib library.

High-level functions like these make it very convenient to use the library when standard configurations and defaults are in play. They are the easiest to use when developing examples or trying to understand the "flow" of an authentication operation without getting overwhelmed by the details.

This makes simple jobs easy and if you need more sophistication and power, you can employ the full power of the CryptoAuthLib object model.

See the Doxygen documentation in [cryptoauthlib/docs](#) for details on the API of the Basic commands. crypto directory - Purpose

This directory contains software implementations of cryptographic functions.

The functions at the base level are wrappers that will point to the final implementations of the software crypto functions.

HAL Directory - Purpose

This directory contains all the Hardware Abstraction Layer (HAL) files used to adapt the upper levels of atca-ng and abstractions to physical hardware.

HAL contains physical implementations for I2C, SWI, SPI, UART and timers for specific hardware platforms.

Include just those HAL files you require based on platform type.

CryptoAuthLib Supported HAL Layers

HAL Layers files are combined into groups. Initial group is generic files that are typically included in a project.

Files are then broken out by uController Family and or Operating System Interface.

Protocol Files	Interface	Files	API	Notes
atca		atca_hal.c/h		For all projects
kit protocol		kit_protocol.c/h		For all Kit Protocol projects
		kit_phy.h		
		hal_i2c_bitbang.c/h	ASF	For all I2C Bitbang projects
		hal_swi_bitbang.c/h	ASF	For all SWI Bitbang projects

Most microcontrollers supported by [Atmel START](#)

have generic drivers depending on the interface.

START Micros	Interface	Files	API	Notes
		hal_timer_start.c	START	Timer implementation
	I2C	hal_i2c_start.c/h	START	
	SWI	swi_uart_start.c/h	START	SWI using UART

AVR Micros	Interface	Files	API	Notes
at90usb1287	I2C	hal_at90usb1287_i2c_asf.c/h	ASF	
		hal_at90usb1287_timer_asf.c	ASF	
	SWI	swi_uart_at90usb1287_asf.c/h	ASF	
xmega_a3bu	I2C	hal_xmega_a3bu_i2c_asf.c/h	ASF	
		hal_xmega_a3bu_timer_asf.c	ASF	
	SWI	swi_uart_xmaga_a3bu_asf.c/h	ASF	

SAM Micros	Interface	Files	API	Notes
sam4s	I2C	hal_sam4s_i2c_asf.c/h	ASF	
		hal_sam4s_timer_asf.c	ASF	

samb11	I2C	hal_samb11_i2c_asf.c/h	ASF	
		hal_samb11_timer_asf.c	ASF	
samd21	I2C	hal_samd21_i2c_asf.c/h	ASF	
		hal_samd21_timer_asf.c	ASF	For all samd21 ASF projects
samd21	I2C	i2c_bitbang_samd21.c/h	ASF	For samd21 I2C bitbang projects
samd21	SWI	swi_bitbang_samd21.c/h	ASF	For samd21 SWI bitbang projects
samd21	SWI	swi_uart_samd21.c/h	ASF	For samd21 SWI uart projects
samg55	I2C	hal_samg55_i2c_asf.c/h	ASF	
		hal_samg55_timer_asf.c	ASF	
samv71	I2C	hal_samv71_i2c_asf.c/h	ASF	
		hal_samv71_timer_asf.c	ASF	

PIC Micros	Interface	Files	API	Notes
pic32mx695f512h	I2C	hal_pic32mx695f512h.c/h	plib.h	For pic32mx695f512h Standalone Mplab projects
		hal_pic32mx695f512h_timer.c	plib.h	For pic32mx695f512h Standalone Mplab projects
PIC32MZ2048	I2C	hal_pic32mz2048efm_i2c.c/h		
		hal_pic32mz2048efm_timer.c		

OS	Interface	Files	API	Notes
MS Windows	kit-cdc	hal_win_kit_cdc.c/h	windows.h	For all windows USB CDC projects
MS Windows	kit-hid	hal_win_kit_hid.c/h	windows.h	For all windows USB HID projects
			setupapi.h	
MS Windows		hal_win_timer.c	windows.h	For all windows projects
Linux	I2C	hal_linux_i2c_userspace.c/h	i2c-dev	
Linux	kit-cdc	hal_linux_kit_cdc.c/h	fopen	For USB Linux CDC projects
Linux	kit-hid	hal_linux_kit_hid.c/h	udev	For USB Linux HID Projects
Linux/Mac		hal_linux_timer.c		For all Linux/Mac projects
All	kit-hid	hal_all_platforms_kit_hidapi.c/h	hidapi	Works for Windows, Linux, and Mac
freeRTOS		hal_freertos.c		freeRTOS common routines

Secure boot using ATECC608A

The SecureBoot command is a new feature on the

[ATECC608A](#) device compared

to earlier CryptoAuthentication devices from Microchip. This feature helps the MCU to identify fraudulent code installed on it. When this feature is implemented, the MCU can send a firmware digest and signature to the ATECC608A. The ATECC608A validates this information (ECDSA verify) and responds to host with a yes or no answer.

The ATECC608A provides options to reduce the firmware verification time by storing the signature or digest after a good full verification (FullStore mode of the SecureBoot command).

- When the ATECC608A stores the digest (SecureBootMode is FullDig), the host only needs to send the firmware digest, which is compared to the stored copy. This skips the comparatively lengthy ECDSA verify, speeding up the secure boot process.
- When the ATECC608A stores the signature (SecureBootMode is FullSig), the host only needs to send the firmware digest, which is verified against the stored signature using ECDSA. This saves time by not needing to send the signature in the command over the bus.

The ATECC608A also provides wire protection features for the SecureBoot command, which can be used to encrypt the digest being sent from the host to the ATECC608A and add a MAC to the verify result coming back to the host so it can't be forced to a success state. This feature makes use of a shared secret between the host and ATECC608A, called the IO protection key.

The secure boot feature can be easily integrated to an existing project. The project should include the following files from the secure_boot folder:

- `secure_boot.c`
- `secure_boot.h`
- `secure_boot_memory.h`
- `io_protection_key.h`

The project should also implement the following platform-specific APIs:

- `secure_boot_init_memory()`
- `secure_boot_read_memory()`
- `secure_boot_deinit_memory()`
- `secure_boot_mark_full_copy_completion()`
- `secure_boot_check_full_copy_completion()`
- `io_protection_get_key()`
- `io_protection_set_key()`

The project can set the secure boot configuration with the following defines:

- `SECURE_BOOT_CONFIGURATION`
- `SECURE_BOOT_DIGEST_ENCRYPT_ENABLED`
- `SECURE_BOOT_UPGRADE_SUPPORT`

The secure boot process is performed by initializing CryptoAuthLib and calling the `secure_boot_process()` function.

Implementation Considerations

- Need to perform SHA256 calculations on the host. CryptoAuthLib provides a software implementation in `lib/crypto/atca_crypto_sw_sha2.c`
- When using the wire protection features:
 - The host needs to be able to generate a nonce (number used once). This is the NumIn parameter to the Nonce command that is sent before the SecureBoot command. The ATECC608A can not be used to generate NumIn, but it should come from a good random or non-repeating source in the host.
 - If the host has any protected internal memory, it should be used to store its copy of the IO protection key.

- Secure boot depends on proper protections of the boot loader code in the host. If the code can be easily changed, then the secure boot process can be easily skipped. Boot loader should ideally be stored in an immutable (unchangeable) location like a boot ROM or write-protected flash.
- Note that these APIs don't provision the ATECC608A. They assume the ATECC608A has already been configured and provisioned with the necessary keys for secure boot.

Examples

For more information about secure boot, please see the example implementation project and documentation at:

https://github.com/MicrochipTech/cryptoauth_usecase_securebootapp directory - Purpose

This directory is for application specific implementation of various use cases.

Methods in this directory provide a simple API to perform potentially complex combinations of calls to the main library or API.

IP Protection with Symmetric Authentication

The IP protection can be easily integrated to the existing projects. The user project should include symmetric_authentication.c & symmetric_authentication.h files which contains the api

- **symmetric_authenticate()** - For Performing the authentication between host & device.

User Considerations

- The user should take care on how the master key should be stored on the MCU side.
- The api's in the file doesn't do the provisioning of the chip and user should take care of the provisioning.

With the provisioned cryptoauthentication device and after doing the cryptoauthlib initialisation, user should only be calling the function symmetric_authenticate() with its necessary parameters for the authentication. The returned authentication status should be used in the application.

Examples

For more information about IP protection and its example project refer [Microchip github](https://github.com/MicrochipTech/cryptoauth_usecase_ip_protection)

TNG Functions

This folder has a number of convenience functions for working with TNG devices (currently ATECC608A-MAHTN-T).

These devices have standard certificates that can be

easily read using the functions in tng_atcacert_client.hCryptoAuthLib - Microchip CryptoAuthentication Library

Introduction

This code base implements an object-oriented C library which supports Microchip CryptoAuth devices. The family of devices supported currently are:

- [ATSHA204A](#)
- [ATECC108A](#)
- [ATECC508A](#)
- [ATECC608A](#)

Online documentation is at <https://microchiptech.github.io/cryptoauthlib/>

Latest software and examples can be found at:

- <http://www.microchip.com/SWLibraryWeb/product.aspx?product=CryptoAuthLib>
- <https://github.com/MicrochipTech/cryptoauthtools>

Prerequisite skills:

- strong C programming and code reading
- Atmel Studio familiarity
- Knowledge of flashing microcontrollers with new code
- Familiarity with Microchip CryptoAuth device functionality

Prerequisite hardware to run CryptoAuthLib examples:

- [ATSAMR21 Xplained Pro](#)
or [ATSAMD21 Xplained Pro](#)
- [CryptoAuth Xplained Pro Extension](#)
or [CryptoAuthentication SOIC Socket Board](#)
to accept SOIC parts

For most development, using socketed top-boards is preferable until your configuration is well tested, then you can commit it to a CryptoAuth Xplained Pro Extension, for example. Keep in mind that once you lock a device, it will not be changeable.

There are two major compiler defines that affect the operation of the library.

- ATCA_NO_POLL can be used to revert to a non-polling mechanism for device responses. Normally responses are polled for after sending a command, giving quicker response times. However, if ATCA_NO_POLL is defined, then the library will simply delay the max execution time of a command before reading the response.
- ATCA_NO_HEAP can be used to remove the use of malloc/free from the main library. This can be helpful for smaller MCUs that don't have a heap implemented. If just using the basic API, then there shouldn't be any code changes required. The lower-level API will no longer use the new/delete functions and the init/release functions should be used directly.

Examples

- Watch [CryptoAuthLib Documents](#) for new examples coming online.
- Node Authentication Example Using Asymmetric PKI is a complete, all-in-one example demonstrating all the stages of crypto authentication starting from provisioning the Crypto Authentication device ATECC608A/ATECC508A with keys and certificates to demonstrating an authentication sequence using asymmetric techniques.
<http://www.microchip.com/SWLibraryWeb/product.aspx?product=CryptoAuthLib>

Release notes

08/30/2019

- Added big-endian architecture support
- Fixes to atcah_gen_dig() and atcah_nonce()

05/17/2019

- Added support for TNG devices (cert transforms, new API)
- atcab_write_pub_key() now works when the data zone is unlocked

03/04/2019

- mbed TLS wrapper added
- Minor bug fixes

01/25/2019

- Python JWT support
- Python configuration structures added
- Restructure of secure boot app

01/04/2019

- Added GCM functions
- Split AES modes into separate files
- Bug fix in SWI START driver

10/25/2018

- Added basic certificate functions to the python wrapper.
- Added Espressif ESP32 I2C driver.
- Made generic Atmel START drivers to support most MCUs in START.
- Added AES-CTR mode functions.
- Python wrapper functions now return single values with AtcaReference.
- Added mutex support to HAL and better support for freeRTOS.

08/17/2018

- Better support for multiple kit protocol devices

07/25/2018

- Clean up python wrapper

07/18/2018

- Added ATCA_NO_HEAP define to remove use of malloc/free.
- Moved PEM functions to their own file in atcacert.

- Added wake retry to accomodate power on self test delay.
- Added ca_cert_def member to atcacert_def_s so cert chains can be traversed as a linked list.

03/29/2018

- Added support for response polling by default, which will make commands return faster (define ATCA_NO_POLL to use old delay method).
- Removed atcats related files as they were of limited value.
- Test framework generates a prompt before locking test configuration.
- Test framework puts device to sleep between tests.
- Fixed mode parameter issue in atcah_gen_key_msg().
- ATECC608A health test error code added.

01/15/2018

- Added AES-128 CBC implementation using AES command
- Added AES-128 CMAC implementation using AES command

11/22/2017

- Added support for FLEXCOM6 on SAMG55 driver

11/17/2017

- Added library support for the ATECC608A device
- Added support for Counter command
- atca_basic functions and tests now split into multiple files based on command
- Added support for multiple base64 encoding rules
- Added support for JSON Web Tokens (jwt)
- Fixed atcab_write_enc() function to encrypt the data even when the device is unlocked
- Fixed atcab_base64encode_() for the extra newline
- Updated atcab_ecdh_enc() to work more consistently

07/01/2017

- Removed assumption of SN[0:1]=0123, SN[8]=EE. SN now needs to be passed in for functions in atca_host and atca_basic functions will now read the config zone for the SN if needed.
- Renamed atcab_gendig_host() to atcab_gendig() since it's not a host function. Removed original atcab_gendig(), which had limited scope.
- Fixed atcah_hmac() for host side HMAC calculations. Added atcab_hmac().
- Removed unnecessary ATCDeviceType parameters from some atca_basic functions.
- Added atcacert_create_csr() to create a signed CSR.
- New HAL implementation for Kit protocol over HID on Linux. Please see the Incorporating CryptoAuthLib in a Linux project using USB HID devices section in this file for more information.
- Added atcacert_write_cert() for writing certificates to the device.
- Added support for dynamic length certificate serial numbers in atcacert.
- Added atcab_write() for lower level write commands.
- Fixed atcah_write_auth_mac(), which had wrong OpCode.
- Added atcab_verify() command for lower level verify commands.
- Added atcab_verify_stored() for verifying data with a stored public key.
- Removed atcab_write_bytes_slot(). Use atcab_write_bytes_zone() instead.
- Modified atcab_write_bytes_zone() and atcab_read_bytes_zone() to specify a

slot

- Added atcab_verify_validate() and atcab_verify_invalidate()
- Improvements to host functions to handle more cases.
- Added atcab_updateextra(), atcab_derive_key()
- Added support for more certificate formats.
- Added general purpose hardware SHA256 functions. See atcab_hw_sha2_256().
- Removed device specific config read/write. Generic now handles both.
- Removed unnecessary response parameter from lock commands.
- Enhanced and added unit tests.
- Encrypted read and write functions now handle keys with SlotConfig.NoMac set
- atcab_cmp_config_zone() handles all devices now.
- Fixed some edge cases in atcab_read_bytes_zone().
- Updated atSHA() to work with all devices.
- Fixed atcacert_get_device_locs() when using stored sn.

01/08/2016

- New HAL implementations for
 - Single Wire interface for SAMD21 / SAMR21
 - SAMV71 I2C HAL implementation
 - XMega A3Bu HAL implementation
- Added atcab_version() method to return current version string of libary to application
- New Bus and Discovery API
 - returns a list of ATCA device configurations for each CryptoAuth device found
 - currently implemented on SAMD21/R21 I2C, SAMV71
 - additional discovery implementations to come
- TLS APIs solidified and documented
- Added missing doxygen documentation for some CryptoAuthLib methods
- Stubs for HAL SPI removed as they are unused for SHA204A and ECC508A support
- bug fixes
- updated atcab_sha() to accept a variable length message that is > 64 bytes and not a multiple of 64 bytes (the SHA block size).
- refactored Cert I/O and Cert Data tests to be smaller
- 'uncrustify' source formatting
- published on GitHub

9/19/2015

- Kit protocol over HID on Windows
- Kit protocol over CDC on Linux
- TLS integration with ATECC508A
- Certificate I/O and reconstruction
- New SHA2 implementation
- Major update to API docs, Doxygen files found in cryptoauthlib/docs
- load cryptoauthlib/docs/index.html with your browser

Host Device Support

CryptoAuthLib will run on a variety of platforms from small micro-controllers to desktop host systems. The current list of hardware abstraction layer support includes:

Rich OS Hosts:

- Linux Kit Protocol over CDC USB
- Linux Kit Protocol over HID USB
- Linux I2C protocol.
- Windows Kit Protocol over CDC USB
- Windows Kit Protocol over HID USB

Microcontrollers:

- SAMD21 (I2C, SWI, and Bit Banging)
- SAMR21 (I2C and SWI)
- SAM4S (I2C)
- SAMV71 (I2C)
- SAMB11 (I2C)
- SAMG55 (I2C)
- AVR XMEGA A3BU (I2C and SWI)
- AVR AT90USB1287 (I2C and SWI)
- PIC32MX695F512H (I2C)

If you have specific microcontrollers or Rich OS platforms you need support for, please contact us through the Microchip portal with your request.

CryptoAuthLib Architecture

See the 'docs' directory of CryptoAuthLib for supporting documentation including architecture diagrams and more detailed usage docs.

The library is structured to support portability to:

- multiple hardware/microcontroller platforms
- multiple environments including bare-metal, Windows, and Linux OS
- multiple chip communication protocols (I2C, SPI, UART, and SWI)

All platform dependencies are contained within the HAL (hardware abstraction layer).

There are three primary object types in CryptoAuthLib:

- Device (ATCADevice)
- Command (ATCACmd)
- Interface (ATCAIface)

ATCADevice is a composite object made up of ATCACmd ATCAIface.

Currently, the vast majority of testing has been performed on:

- ATSAMR21 Xplained Pro
- ATSAMD21 Xplained Pro
- ATSAMV71 Xplained Pro
- Windows (kit protocol HID)

These host containers implement a host test environment and test console to exercise tests. They presume that a CryptoAuth Xplained Pro or other I2C socket for an ATECC608A/ATECC508A/ATECC108A/ATSHA204A are

connected to the I2C pins of the host Xplained Pro development board or in the case of windows is using a HID connection to an ATCK101 or ATCK590.

The unit tests and basic tests exercise the core datasheet commands of the device as well as the more convenient, basic API methods.

If you need an example of how to use a command, these hosts and tests are a good place to reference.

Object Architecture

Even though this is a C library, it follows object-oriented design patterns.

An object is minimally defined to be data and the actions which operate on that data.

Each CryptoAuth device is a composite object, a structure which includes the command table (list of commands) which are valid for the device, and the data used to hold the state of that device.

ATCADevice is the object which represents the Microchip CryptAuth device

ATCACmd is the object which represents the valid methods of the Device.

ATCAInterface is the physical interface object (I2C or SWI instance).

Currently, each Device may have a single OATCAInterface.

ATCADevice represents an ATSHA or ATECC family device.

In order to add new protocol support for a platform, you provide a HAL (hardware abstraction layer) C file for the protocol and target. In your project's IDE or Makefile, you select which HAL support you need for the hardware configuration. Generally, there are separate files for each protocol and platform combination - (ie: samd21_i2c_asf.c would target SAMD21 MCUs with I2C using the ASF low-level driver support.)

Directory Structure

```
docs - AppNotes and Doxygen HTML documentation for the library API. Load "docs/html/index.html" in your browser
lib - primary library source code
lib/atcacert - certificate data and i/o methods
lib/basic - the Basic API way to access the core classes
lib/crypto - Software crypto implementations (primarily SHA1 and SHA256)
lib/hal - hardware abstraction layer code for supporting specific platforms
lib/host - support functions for common host-side calculations
lib/jwt - json web token functions
test - unit tests. See test/cmd-processor.c for main() implementation.
```

For production code, `test` directories should be excluded by not compiling it into a project, so it is up to the developer to include or not as needed. Test code adds significant bulk to an application - it's not intended to be included [in](#) production code.

Tests

There is a set of unit tests found in the test directory which will at least partially demonstrate the use of the objects. Some tests may depend upon a certain device being configured in a certain way and may not work for all devices or specific configurations of the device.

The test/cmd-processor.c file contains a main() function for running the tests. It implements a command-line interface. Typing help will bring up the list of commands available.

One first selects a device type, with one of the following commands:

- 204 (ATSHA204A)
- 108 (ATECC108A)
- 508 (ATECC508A)
- 608 (ATECC608A)

From there the following unit test sweets are available:

- unit (test command builder functions)
- basic (test basic API functions)
- cio (test certification i/o functions)
- cd (test certificate data functions)
- util (test utility functions)
- crypto (test software crypto functions)

Unit tests available depend on the lock level of the device. The unit tests won't lock the config or data zones automatically to allow retesting at desired lock levels. Therefore, some commands will need to be repeated after locking to exercise all available tests.

Starting from a blank device, the sequence of commands to exercise all unit tests is:

```
unit
basic
lockcfg
unit
basic
lockdata
unit
basic
cio
cd
util
crypto
```

Using CryptoAuthLib (Microchip CryptoAuth Library)

Using a new library is often easier when you can load an example and see how it works. We've provided examples in the form of "host containers" which are host projects that incorporate CryptoAuthLib and target various processors or communication APIs.

We maintain host test containers for each of the HAL layers we support. We've published the host container for SAMD21 which demonstrates a simple console interface to invoke test runners.

Look for SAMD21 Unit Tests CryptoAuthLib at
<http://www.microchip.com/SWLibraryWeb/product.aspx?product=CryptoAuthLib>

The best way to learn how to use CryptoAuthLib is to study the host test projects that exercise the library and ATECC and ATSHA devices.

New examples will be forthcoming as the software matures. Continue checking

the [CryptoAuthentication](#)

web page for new updates.

Using Git to Incorporate CryptoAuthLib as a Submodule

You can include this project in your own project under git.

Using CryptoAuthLib as a git submodule, you can maintain your application separately from CryptoAuthLib.

If your project is already in git but you haven't yet intergrated

CryptoAuthLib, change to the directory where you want to put CryptoAuthLib

```
git submodule add -b master <giturl to CryptoAuthLib>
```

This adds CryptoAuthLib as a subdirectory and separate git repo within your own project. Changes and commits to your project vs CryptoAuthLib will remain separated into each respective repository.

If there is a project you want to checkout that already incorporates CryptoAuthLib as a submodule if you clone the repo that incorporates CryptoAuthLib, after cloning, you'll still need to fill out the CryptoAuthLib submodule after cloning:

```
git submodule init  
git submodule update --remote  
cd cryptoauthlib  
git checkout master
```

Now that CryptoAuthLib is a full-fledged submodule in your git project, in order to easily add it to your project within Atmel Studio, please see this [tip](#)

Incorporating CryptoAuthLib in a project

1. In your Makefile or IDE, choose the HAL support you need from the HAL directory and exclude other HAL files from your project.
2. For I2C interfaces, define the symbol ATCA_HAL_I2C in your compiler's symbol definitions. This will hook up the CryptoAuthLib interface class with your HAL implementation of I2C methods.
3. HAL implementations for CDC and HID interfaces to the ATCK101 are also included for use with Windows or Linux versions of the test host.

Incorporating CryptoAuthLib in a Linux project using USB HID devices

The Linux HID HAL files use the Linux udev development software package.

To install the udev development package under Ubuntu Linux, please type the following command at the terminal window:

```
sudo apt-get install libudev-dev
```

This adds the udev development development software package to the Ubuntu Linux installation.

The Linux HID HAL files also require a udev rule to be added to change the

permissions of the USB HID Devices. Please add a new udev rule for the Microchip CryptoAuth USB devices.

```
cd /etc/udev/rules.d  
sudo touch mchp-cryptoauth.rules
```

Edit the mchp-cryptoauth.rules file and add the following line to the file:

```
SUBSYSTEM=="hidraw", ATTRS{idVendor}=="03eb", ATTRS{idProduct}=="2312", MODE=="0666"  
`# ATCA support for Mongoose OS
```

This is a HAL implementation for the Atmel/Microchip CryptoAuthLib. It translates ATCA interface calls into MGOS API calls. Currently only I2C is implemented.

Embedded JavaScript engine

This library brings an [mJS: restricted JavaScript-like Engine](<https://github.com/cesanta/mjs>).

Apart from adding the mJS itself, the library creates a global instance of it (available via `mgos_mjs_get_global()`), and also brings a number of mgos-specific API files, see `fs` directory.

How to contribute

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](<https://cla.developers.google.com/about/google-individual>).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](<https://cla.developers.google.com/about/google-corporate>).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
1. Follow the normal process of [forking](<https://help.github.com/articles/fork-a-repo>) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
1. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
1. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](<https://github.com/mbrukman/autogen>) tool.
1. Do your best to have [well-formed commit messages](<http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html>) for each change. This provides consistency throughout the project, and

ensures that commit messages are able to be formatted properly by various git tools.

- Finally, push the commits to your fork and submit a [pull request](<https://help.github.com/articles/creating-a-pull-request>).
VEML6075 I2C Driver

A Mongoose library for Vishay Technologies UVA/UVB intensity sensor.

Implementation details

The VEML6075 sensor can be configured in continuous or one-shot measurement modes, and upon each measurement returns the 16-bit raw counters for UVA, UVB, visual and IR bands. Using conversion coefficients, the UVIndex can be returned using these data points.

An accurate VEML6075 UVI sensing system requires visible and infrared noise compensation and a teflon diffusor for cosine angular response correction. The UVI formulas and related UVI formula coefficients are discussed here in detail.

This library implements the application [notes](<https://www.vishay.com/docs/84339/designingveml6075.pdf>)

Limitations

The sensor measures for a configurable interval and integrates the total light received for each band. Although its sensitivity can easily be adjusted with selecting the proper integration times (from 50ms through to 800ms), the algorithm provided in the Vishay paper is only applicable to 100ms integration times. Therefore, the library currently does not support other integration times.

Example application

An example program using a timer to read data from the sensor every 5 seconds:

include "mgos.h"

include "mgos_veml6075.h"

```
static struct mgos_veml6075 *s_veml6075;

static void timer_cb(void *user_data) {
    float uva, uvb, uvindex;

    uva=mgos_veml6075_getUVA(s_veml6075);
    uvb=mgos_veml6075_getUVB(s_veml6075);
    uvindex=mgos_veml6075_getUVIndex(s_veml6075);

    LOG(LL_INFO, ("VEML6075 uva=%.1f uvb=%.1f uvindex=%.2f", uva, uvb, uvindex));

    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    s_veml6075=mgos_veml6075_create(0x10); // Default I2C address
    if (s_veml6075) {
        mgos_set_timer(5000, true, timer_cb, NULL);
    } else {
        LOG(LL_ERROR, ("Could not initialize sensor"));
    }
    return MGOS_APP_INIT_SUCCESS;
}
```

}

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Arduino Adafruit HTU21D(D) library for Mongoose OS

This is a port of the [Adafruit Arduino driver for the HTU21D(F) sensor](https://github.com/adafruit/Adafruit-HTU21DF_Library) ported to run on the [Mongoose OS](<https://mongoose-os.com/>).

Usage is extremely simple....

in _**mos.yml**_, add to **libs:** section,

```
` - origin: https://github.com/mongoose-os-libs/arduino-adafruit-htu21df`
```

in your _**init.js**_, add something like the following,

```
```javascript
load('api_arduino_htu21df.js');
```

and

```
// Initialize Adafruit-HTU21DF library
let htu = Adafruit-HTU21DF.create();
htu.begin();
let htuGetData = function () {
 print('HTU21D: T: ', htu.readTemperature() , 'C RH: ', htu.readHumidity(), '%');
};
let htuTimer = Timer.set(10000 /* milliseconds */, true /* repeat */, htuGetData, null);
```

to use the library.

Enjoy!

## How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

## Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it

again.

## Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

## MAX7219/MAX7221 SPI Driver

A Mongoose library for MAX7219, an 8-digit LED Display Driver.

### Implementation details

The chip is straight forwardly implemented, and allows for daisy chaining multiple devices (by tying `DOUT` of the first chip to `DIN` of the second chip, while connecting `CS`, `CLK`, `VCC` and `GND` to all chips).

After initialization, bytes can be written to so called `digits`, which are the individual lines of an 8x8 LED display, or the actual digits of a 7-segment display.

**NOTE:** The MAX7219 is a 5V device, and as such level shifters are necessary between the `MOSI`, `SCLK` and `CS` pins of the MCU and the pins on the chip(s). It may work without, but it's not recommended!

### API Description

Create the object with `mgos_max7219_create()`, and if there are multiple devices connected, set the correct number with `mgos_max7219_set_num_devices()`. When you're done using the display(s), call `mgos_max7219_destroy()` to turn them off and clean them up.

**`mgos_max7219_set_intensity()`** Intensity of all displays will be set, where the argument is between 0 (very dim) to 15 (very bright).

**mgos\_max7219\_set\_mode()** Setting the mode is done with a boolean mode argument. If true is specified, all connected devices are set into what is called Code B decoding, in which the values map internally to segments of a 7-segment display, for example writing 0 will light up the segments needed to draw a 0 on a 7-segment display. There are only 15 values in this system, see the header file for details. The mode is initialized as false, in other words do not use Code B, and now each bit on the 8-bit values correspond to one LED, typically on an 8x8 LED display.

Writes can be performed in three ways:

**mgos\_max7219\_write\_raw()** writes an 8-bit value to a certain line (digit) of a certain device (deviceno). This will only work if the mode was set to false.

**mgos\_max7219\_write\_digit()** writes a number to a certain position (digit) of a certain device (deviceno). This will only work if the mode was set to true.

**mgos\_max7219\_write\_line()** writes an array of bytes, which must be of exactly num\_devices length, to the given line (digit) of all connected displays.

## Example application

```

#include "mgos.h"
<p class="mume-header" id="include-mgosh-28"></p>

#include "mgos_config.h"
<p class="mume-header" id="include-mgos_configh-6"></p>

#include "mgos_max7219.h"
<p class="mume-header" id="include-mgos_max7219h"></p>

const uint8_t program[] = {
 0, 153, 90, 60, 24, 24, 36, 66, 5,
 0, 24, 24, 255, 24, 24, 36, 66, 5,
 0, 24, 24, 60, 90, 153, 36, 66, 5,
 0, 24, 24, 255, 24, 24, 36, 66, 5,
};

void timer_cb(void *data) {
 struct mgos_max7219 *d = (struct mgos_max7219 *)data;
 static uint8_t waiting = 0;
 static uint16_t position = 0;

 if (!d) return;

 if (waiting > 0) {
 waiting--;
 return;
 }

 waiting = program[position * 9 + 8];
 LOG(LL_INFO, ("Writing program position %u, then waiting %.1f seconds", position, 0.1 * waiting));

 for (int i = 0; i < 8; i++) {
 mgos_max7219_write_raw(d, 0, i, program[position * 9 + i]);
 mgos_max7219_write_raw(d, 1, i, ~(program[position * 9 + i]));
 }

 position++;
 if (position == sizeof(program) / 9) position = 0;
}

enum mgos_app_init_result mgos_app_init(void) {
 struct mgos_max7219 *d = NULL;

 if (!(d = mgos_max7219_create(mgos_spi_get_global(), mgos_sys_config_get_max7219_cs_index())))
 LOG(LL_ERROR, ("Could not create MAX7219 display"));
 return MGOS_APP_INIT_ERROR;
}
 mgos_max7219_set_num_devices(d, 2);
 mgos_set_timer(100, true, timer_cb, d);

 return MGOS_APP_INIT_SUCCESS;
}

```

## Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

## A Texas Instruments TMP006 temperature sensor

### Overview

This is a library for the Texas Instruments TMP006 temperature sensor.

See <http://www.ti.com/product/TMP006> for more information about the hardware.

# Support library for the device management dashboard

## Overview

This library provides support for device management dashboard.

# Implementation of Mongoose OS RPC over UDP

## Overview

Frames are sent as datagrams. Fragmentation is not supported, so 1 frame = 1 datagram.

Responses are supported, but because UDP is not reliable, it is best suited for notifications.

## Listener configuration

Set `rpc.udp.listen_addr` to `udp://0.0.0.0:1234` to listen on port 1234.

## Outbound channel configuration

To use UDP for outbound RPC, set `rpc.udp.dst_addr` to the address of the server / peer, e.g. `udp://192.168.1.23:1234`.

## mos tool support

`mos` supports UDP transport via the `udp://` port scheme, e.g.:

```
$ mos call --port udp://192.168.11.5:1234/ Sys.GetInfo
```

## How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

## Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted

one (even if it was for a different project), you probably don't need to do it again.

## Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

## Acknowledgements

This code was inspired by existing work by the following entities:

### Adafruit Industries

Portions of this code are taken from Adafruit's STMPE610 library:

- The initialization sequence and defines in `include/mgos_stmpe610.h`

Adafruit licensed their code under the [MIT License](#).

## STMP610E

Mongoose native SPI driver for STMP610E

## Introduction

STMPE610 is a chip that drives a resistive touchscreen. It measures pressure on a glass plate which has an X and Y resistor samples the pressure put on the screen (typically by pressing it).

## API

This driver initializes the chip and allows the user to register a callback, which will be called each time the driver senses a touch (`TOUCH_DOWN`) or

a release (`TOUCH_UP`). The callback function will be given the `X` and `Y` coordinates, together with an average pressure (`Z`). For `TOUCH_DOWN` events, the `length` field is set to 1. For `TOUCH_UP` events, the `length` field is set to the amount of samples read, between 1 and 128.

The `X` and `Y` coordinates as read by the screen are between [0..4095), so a mapping function is applied by `mgos_stmpe610_set_dimensions()`. For screens that are rotated, `mgos_stmpe610_set_rotation()` sets the mapping function to return them in the correct orientation.

## Details

The chip uses an interrupt pin to signal the micro controller that it has something to say. The driver sets up a GPIO pin with a pullup, which the chip pulls to ground to initiate the interrupt. Mongoose OS will receive this and issue a software callback (so no hardware interrupts are used), and the driver will clear interrupts on the STMPE610 after the user supplied handler is called.

Users can query the driver to see if the chip is currently registering a touch by calling `mgos_stmpe610_is_touching()` which will return `true` if the user is currently touching the screen.

If the user touches the screen very briefly (only one sample cycle of the driver), the callback will be called with `TOUCH_DOWN`, but no `TOUCH_UP` event will be generated. To work around this, the driver sets a 100ms timer upon `TOUCH_DOWN`, and if the screen is not being touched when the timer expires, a gratuitous `TOUCH_UP` event is sent with `length` set to 1.

## Example Application

### mos.yml

The driver uses the Mongoose native SPI driver. It is configured by setting up the `MOSI`, `MISO`, `SCLK` pins and assinging one of the three available `CS` positions, in this example (which was taken from the Huzzah32 ESP32 microcontroller), we are going to use `CS0`:

```
config_schema:
 - ["spi.enable", true]
 - ["spi.cs0_gpio", 32] # The STMPE610 CS pin
 - ["spi.cs1_gpio", -1]
 - ["spi.cs2_gpio", -1]
 - ["spi.mosi_gpio", 18]
 - ["spi.miso_gpio", 19]
 - ["spi.sclk_gpio", 5]
 - ["stmpe610.irq_pin", 23]
 - ["stmpe610.cs_index", 0] # Use spi.cs0_gpio
```

## Application

```

include "mgos.h"
<p class="mume-header" id="include-mgosh-29"></p>

include "mgos_stmpe610.h"
<p class="mume-header" id="include-mgos_stmpe610h"></p>

static void touch_handler(struct mgos_stmpe610_event_data *ed) {
 if (!ed) return;

 LOG(LL_INFO, ("Touch %s at (%d,%d) pressure=%d, length=%d",
 ed->direction==TOUCH_UP?"UP":"DOWN", ed->x, ed->y, ed->z, ed->length));
}

enum mgos_app_init_result mgos_app_init(void) {
 mgos_stmpe610_set_handler(touch_handler);
 mgos_stmpe610_set_rotation(STMPE610_LANDSCAPE);
 mgos_stmpe610_set_dimensions(320, 240);

 return MGOS_APP_INIT_SUCCESS;
}

```

# Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

## A short guide to use fontconvert.c to create your own fonts using MinGW.

### STEP 1: INSTALL MinGW

Install MinGW (Minimalist GNU for Windows) from [MinGW.org](http://MinGW.org).

Please read carefully the instructions found on [Getting started page](#).

I suggest installing with the "Graphical User Interface Installer".

To complete your initial installation you should further install some "packages".

For our purpose you should only install the "Basic Setup" packages.

To do that:

1. Open the MinGW Installation Manager
2. From the left panel click "Basic Setup".
3. On the right panel choose "mingw32-base", "mingw-gcc-g++", "mingw-gcc-objc" and "msys-base" and click "Mark for installation"
4. From the Menu click "Installation" and then "Apply changes". In the pop-up window select "Apply".

### STEP 2: INSTALL Freetype Library

To read about the freetype project visit [freetype.org](http://freetype.org).

To Download the latest version of freetype go to [download page](#)

and choose "freetype-2.7.tar.gz" file (or a newer version if available).

To avoid long cd commands later in the command prompt, I suggest you unzip the file in the C:\ directory.

(I also renamed the folder to "ft27")

Before you build the library it's good to read these articles:

- [Using MSYS with MinGW](#)
- [Installation and Use of Supplementary Libraries with MinGW](#)
- [Include Path](#)

Inside the unzipped folder there is another folder named "docs". Open it and read the INSTALL.UNIX (using notepad).

Pay attention to paragraph 3 (Build and Install the Library). So, let's begin the installation.

To give the appropriate commands we will use the MSYS command prompt (not cmd.exe of windows) which is UNIX like.

Follow the path C:\MinGW\msys\1.0 and double click "msys.bat". The command prompt environment appears.

Enter "ft27" directory using the cd commands:

```
cd /c
cd ft27
```

and then type one by one the commands:

```
./configure --prefix=/mingw
make
make install
```

Once you're finished, go inside "C:\MinGW\include" and there should be a new folder named "freetype2".

That, hopefully, means that you have installed the library correctly !!

### STEP 3: Build fontconvert.c

Before proceeding I suggest you make a copy of Adafruit\_GFX\_library folder in C:\ directory.

Then, inside "fontconvert" folder open the "makefile" with an editor ( I used notepad++).

Change the commands so in the end the program looks like :

```
all: fontconvert

CC = gcc
CFLAGS = -Wall -I c:/mingw/include/freetype2
LIBS = -lfreetype

fontconvert: fontconvert.c
 $(CC) $(CFLAGS) $< $(LIBS) -o $@

clean:
 rm -f fontconvert
```

Go back in the command prompt and with a cd command enter the fontconvert directory.

```
cd /c/adafruit_gfx_library/fontconvert
```

Give the command:

```
make
```

This command will, eventually, create a "fontconvert.exe" file inside fontconvert directory.

### STEP 4: Create your own font header files

Now that you have an executable file, you can use it to create your own fonts to work with Adafruit GFX lib.

So, if we suppose that you already have a .ttf file with your favorite fonts, jump to the command prompt and type:

```
./fontconvert yourfonts.ttf 9 > yourfonts9pt7b.h
```

You can read more details at: [learn.adafruit](#).

Taraaaaaammm !! you've just created your new font header file. Put it inside the "Fonts" folder, grab a cup of coffee and start playing with your Arduino (or whatever else ....)+ display module project.

## Cesanta note

# Adafruit GFX Library

This is the core graphics library for all our displays, providing a common set of graphics primitives (points, lines, circles, etc.). It needs to be paired with a hardware-specific library for each display device we carry (to handle the lower-level functions).

Adafruit invests time and resources providing this open source code, please support Adafruit and open-source hardware by purchasing products from Adafruit!

Written by Limor Fried/Ladyada for Adafruit Industries.

BSD license, check license.txt for more information.

All text above must be included in any redistribution.

Recent Arduino IDE releases include the Library Manager for easy installation. Otherwise, to download, click the DOWNLOAD ZIP button, uncompress and rename the uncompressed folder Adafruit\_GFX. Confirm that the Adafruit\_GFX folder contains Adafruit\_GFX.cpp and Adafruit\_GFX.h. Place the Adafruit\_GFX library folder your /Libraries/ folder. You may need to create the Libraries subfolder if its your first library. Restart the IDE.

## Useful Resources

- **Image2Code:** This is a handy Java GUI utility to convert a BMP file into the array code necessary to display the image with the drawBitmap function. Check out the code at ehubin's GitHub repository: <https://github.com/ehubin/Adafruit-GFX-Library/tree/master/Img2Code>
- **drawXBitmap function:** You can use the GIMP photo editor to save a .xbm file and use the array saved in the file to draw a bitmap with the drawXBitmap function. See the pull request here for more details: <https://github.com/adafruit/Adafruit-GFX-Library/pull/31>
- 'Fonts' folder contains bitmap fonts for use with recent (1.1 and later) Adafruit\_GFX. To use a font in your Arduino sketch, #include the corresponding .h file and pass address of GFXfont struct to setFont(). Pass NULL to revert to 'classic' fixed-space bitmap font.
- 'fontconvert' folder contains a command-line tool for converting TTF fonts to Adafruit\_GFX .h format.

## Arduino Adafruit GFX library for Mongoose OS

## RPC support for Google Cloud Platform

### Device support

This library provides RPC support over the mechanisms provided by the Google IoT Core platform

- Requests should be sent to the device as **commands**.  
This library listens to commands on a particular subfolder (default: `rpc`) and parses the payload as RPC frames.
- Responses are published as telemetry events, to the subfolder that is equal to the `dst` of the response (and thus `src` in the original request).  
So, if the caller wishes to receive response on subfolder `foo`, they should specify `src: \"foo\"` in their request.

It should be noted that responses from all the devices that use the same response subfolder end up in the same topic and subscription.

All the subscribers will receive messages posted here by all the devices. Subscribers need to take care to not ack messages that are not intended for them (match request ID).

While acceptable for light use, this will cause unnecessary churn if multiple devices are communicating at the same time.

In this case users should set up use multiple different response subfolders.

To receive responses, caller must set up the necessary PubSub topics and/ subscription and ensure that the desired subfolder is plumbed to the correct topic.

See [here](#) for detailed description.

## Mos tool support

mos supports GCP RPC mechanism through the gcp "port" type:

```
$ mos --port gcp://project/region/registry/device call Sys.GetInfo
{
 "app": "demo-c",
 "fw_version": "1.0",
 "fw_id": "20190123-121047/2.10.0-244-g81233d5ed-dirty-mos8",
 "mac": "1AFE34A5930F",
 "arch": "esp8266",
 "uptime": 5248,
 "ram_size": 51864,
 "ram_free": 37072,
 "ram_min_free": 21928,
 "fs_size": 233681,
 "fs_free": 153612,
 "wifi": {
 "sta_ip": "192.168.11.25",
 "ap_ip": "",
 "status": "got ip",
 "ssid": "Over_9000_Internets"
 }
}
```

Use your own project, region, registry, and device.

By default, mos uses rpc subfolder for telemetry events and subscribes to the rpc topic.

You will need to create the topic and forward the rpc subfolder to it.

Or allow mos to set up all the necessary plumbing byt adding --gcp-create-topic (Note: config changes to GCP take some time to propagate, so the first reuquest may fail).

You can customize subfolder names via query parameters (e.g. mos --port=gcp://project/region/registry/device?sub=rpc1&respsf=rpc1):

- sub - subscription name. Default is rpc.
- topic - topic name. Default is rpc.
- reqsf - request subfolder. This needs to match rpc.gcp.subfolder on the device. Default is rpc.
- respsf - response subfolder. Default is rpc.
  - As described above, there is inherent contention between subscribers when the same response subfolder is used for responses to requests by multiple callers.  
If the volume of churn (Nacks) becomes a problem, and until Google implements [this](#) or [this](#), the workaround is to set up several response subfolders and shard responses between them.

## Library for modular scrolling LED matrix text displays

[Version 1.0 Video](#)

[Version 2.0 Video](#)

[Library Documentation](#)

Parola is a modular scrolling text display using MAX7219 or MAX7221 LED matrix display controllers using Arduino. The display is made up of any number of identical modules that are plugged together to create a wider/longer display.

- Text left, right or center justification in the display
- Text scrolling, entry and exit effects

- Control display parameters and animation speed
- Support for hardware SPI interface
- Multiple virtual displays (zones) in each string of LED modules
- User defined fonts and/or individual characters substitutions
- Support for double height displays

The aim was to create a 'lego-like' LED matrix display, using standard 8x8 LED matrices. The software supports this flexibility through a scalable approach that only requires the definition of the number of modules to adapt existing software to a new configuration.

The Parola software has a dependency on the [MD\\_MAX72xx Arduino library](#) which implements hardware functions of the LED matrix. The library needs to be configured for the type of matrices being used - please refer to the hardware section of documentation for the [MD\\_MAX72xx library](#).

Parola discussion on the [Arduino forum](#) and kits available from [ElectroDragon](#).

Additional information also at [my blog](#).

## How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

## Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

## Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.

4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

# Mongoose OS IMU Library

This library provides a simple API that describes inertial measurement units. It implements various popular I2C and SPI IMUs behind that API. Callers of the library can rely on the API returning consistently typed data regardless of the choice of sensor.

## Anatomy

This library provides an API to expose various implementations of gyroscopes, accelerometers, and magnetometers. It offers a suitable abstraction that presents the sensor data in a consistent way, so application authors need not worry about the implementation details of the sensors.

Users create a `struct mgos_imu` object, to which they add sensors (gyroscope, accelerometer and magnetometer) either via I2C or SPI. Some chips have all three sensor types, some have only two (typically accelerometer and gyroscope) and some have only one type. After successfully adding the sensors to the `mgos_imu` object, reads can be performed, mostly by performing `mgos_imu_*_get()` calls.

All implementations offer the calls described below in the [IMU API](#). Some implementations offer chip-specific addendums, mostly setting sensitivity and range options.

## IMU API

### IMU primitives

`struct mgos_imu *mgos_imu_create()` -- This call creates a new (opaque) object which represents the IMU device. Upon success, a pointer to the object will be returned. If the creation fails, NULL is returned.

`void mgos_imu_destroy()` -- This cleans up all resources associated with the IMU device. The caller passes a pointer to the object pointer. If any sensors are attached (gyroscope, accelerometer, magnetometer), they will be destroyed in turn using `mgos_imu_*_destroy()` calls.

`bool mgos_imu_read()` -- This call schedules all sensors attached to the IMU to be read from. It is not generally necessary to call this method directly, as the `mgos_imu_*_get()` calls internally schedule reads from the sensors as well.

`bool mgos_imu_accelerometer_present()` -- This returns `true` if the IMU has an

attached accelerometer sensor, or `false` otherwise.

`bool mgos_imu_gyroscope_present()` -- This returns `true` if the IMU has an attached gyroscope sensor, or `false` otherwise.

`bool mgos_imu_magnetometer_present()` -- This returns `true` if the IMU has an attached magnetometer, or `false` otherwise.

## IMU Sensor primitives

For each of **accelerometer**, **gyroscope** and **magnetometer**, the following primitives exist:

`bool mgos_imu_*_create_i2c()` -- This attaches a sensor to the IMU based on the `type` enum given, using the `i2c` bus and specified `i2caddr` address. The function will return `true` upon success and `false` in case either detection of the sensor, or creation of it failed.

`bool mgos_imu_*_create_spi()` -- This attaches a sensor to the IMU based on the `type` enum given, using the `spi` bus and a specified cable select in `cs_gpio`. The function will return `true` upon success and `false` in case either detection of the sensor or creation of it failed.

`bool mgos_imu_*_destroy()` -- This detaches a sensor from the IMU if it exists. It takes care of cleaning up all resources associated with the sensor, and detaches it from the `i2c` or `spi` bus. The higher level `mgos_imu_destroy()` call uses these lower level calls to clean up sensors as well.

`bool mgos_imu_*_get()` -- This call returns sensor data after polling the sensor for the data. It returns `true` if the read succeeded, in which case the floats pointed to by `x`, `y` and `z` are filled in. If an error occurred, `false` is returned and the contents of `x`, `y` and `z` are unmodified. Note the units of the return values:

- **magnetometer** returns units of Gauss.
- **accelerometer** returns units of G.
- **gyroscope** returns units of degrees per second.

`const char *mgos_imu_*_get_name()` -- This returns a symbolic name of the attached sensor, which is guaranteed to be less than or equal to 10 characters and always exist. If there is no sensor of this type attached, `VOID` will be returned. If the sensor is not known, `UNKNOWN` will be returned. Otherwise, the chip manufacturer / type will be returned, for example `MPU9250` or `ADXL345` or `MAG3110`.

`bool mgos_imu_*_set_orientation()` and `bool mgos_imu_*_get_orientation()` -- Often times a 9DOF sensor will have multiple chips, whose axes do not line up correctly. Even within a single chip the accelerometer, gyroscope and magnetometer axes might not line up (for an example of this, see [MPU9250 chapter 9.1](#)).

To ensure that the `x`, `y`, and `z` axes on all sensors are pointed in the same direction, we can set the orientation on the gyroscope and magnetometer. See `mgos_imu.h` for more details and an example of how to do this.

## Supported devices

### Accelerometer

- MPU9250 and MPU9255
- ADXL345
- LSM303D and LSM303DLM
- MMA8451
- LSM9DS1
- LSM6DSL
- MPU6000 and MPU6050
- ICM20948

## Gyroscope

- MPU9250 and MPU9255
- L3GD20 and L3GD20H
- ITG3205
- LSM9DS1
- LSM6DSL
- MPU6000 and MPU6050
- ICM20948

## Magnetometer

- MAG3110
- AK8963 (as found in MPU9250/MPU9255)
- AK8975
- LSM303D and LSM303DLM
- HMC5883L
- LSM9DS1
- ICM20948

## Adding devices

This is a fairly straight forward process, taking the `ADXL345` accelerometer as example:

1. Add a new driver header and C file, say `src/mgos_imu_adxl345.[ch]`. The header file exposes all of the `#define`'s for registers and constants.
2. In the header file, declare a `detect`, `create`, `destroy` and `read` function.
  - `bool mgos_imu_adxl345_detect()` -- this function optionally attempts to detect the chip, often times by reading a register or set of registers which uniquely identifies it. Not all chips can actually be detected, so it's OK to not define this function at all.
  - `bool mgos_imu_adxl345_create()` -- this function has to perform the initialization of the chip, typically by setting the right registers and possibly creating a driver-specific memory structure (for, say, coefficients or some such). If used, that memory structure is attached to the `user_data` pointer, and if so, the implementation of the `_destroy()` function must clean up and free this memory again.
  - `bool mgos_imu_adxl345_read()` -- this function performs the chip specific read functionality. This will be called whenever the user asks for data, either by calling `mgos_imu_read()` or by calling `mgos_imu_*_get()`.
  - `bool mgos_imu_adxl345_destroy()` -- this function deinitializes the chip, and optionally clears and frees the driver-specific memory structure in `user_data`. Not all chips need additional memory

structures or deinitialization code, in which case it's OK to not define this function at all.

3. Implement the `detect`, `create`, `destroy` and `read` functions in the source code file `src/mglos_imu_adxl345.c`.
4. Add the device to one of the enum `mgos_imu_*_type` in `include/mglos_imu.h`. In the example case of adxl345 (which is an accelerometer), add it to `enum mgos_imu_acc_type`.
5. Add a string version of this to function `mgos_imu_*_get_name()` so that callers can resolve the sensor to a human readable format. Make sure that the string name is not greater than 10 characters.
6. Add the type to the `switch()` in `mgos_imu_*_create_i2c()` (or `_spi()`) functions.
7. Update this document to add the driver to the list of supported drivers.
8. Test code on a working sample, and send a PR using the guidelines laid out in [contributing](#).

It is important to note a few implementation rules when adding drivers:

- New drivers MUST NOT change any semantics of the abstraction layer (`mgos_imu` and `mgos_imu_*` members) nor the `mgos_imu_*_get()` functions.
- Implementations of drivers MUST provide bias, scaling and other normalization in the driver itself. What this means, in practice, is that the correct units must be produced (`m/s/s`, `deg/s` and `uT`).
- Pull Requests MUST NOT mix driver and abstraction changes. Separate them.
- Changes to the abstraction layer MUST be proven to work on all existing drivers, and will generally be scrutinized.

## Example driver (AK8975)

Here's an example, for the magnetometer chip AK8975, showing a set of commits for each of the steps above (and honoring the driver implementation rules).

1. Add `src/mglos_imu_ak8975.[ch]` [commit](#)
2. Extend `enum mgos_imu_mag_type` in `include/mglos_imu.h` [commit](#)
3. Add to `mgos_imu_magnetometer_get_name()` in `src/mglos_imu_magnetometer.c` [commit](#)
4. Add to `mgos_imu_magnetometer_create_i2c()` in `src/mglos_imu_magnetometer.c` [commit](#)

## Example Code

An example program that creates an IMU of type `MPU9250` (which has an accelerometer, a gyroscope and a magnetometer all in one tiny package):

```

#include "mgos.h"
<p class="mume-header" id="include-mgosh-30"></p>

#include "mgos_i2c.h"
<p class="mume-header" id="include-mgos_i2ch-5"></p>

#include "mgos_imu.h"
<p class="mume-header" id="include-mgos_imuh"></p>

static void imu_cb(void *user_data) {
 struct mgos_imu *imu = (struct mgos_imu *)user_data;
 float ax, ay, az;
 float gx, gy, gz;
 float mx, my, mz;

 if (!imu) return;

 if (mgos_imu_accelerometer_get(imu, &ax, &ay, &az))
 LOG(LL_INFO, ("type=%-10s Accel X=%.2f Y=%.2f Z=%.2f",
 mgos_imu_accelerometer_get_name(imu), ax, ay, az));
 if (mgos_imu_gyroscope_get(imu, &gx, &gy, &gz))
 LOG(LL_INFO, ("type=%-10s Gyro X=%.2f Y=%.2f Z=%.2f",
 mgos_imu_gyroscope_get_name(imu), gx, gy, gz));
 if (mgos_imu_magnetometer_get(imu, &mx, &my, &mz))
 LOG(LL_INFO, ("type=%-10s Mag X=%.2f Y=%.2f Z=%.2f",
 mgos_imu_magnetometer_get_name(imu), mx, my, mz));
}

enum mgos_app_init_result mgos_app_init(void) {
 struct mgos_i2c *i2c = mgos_i2c_get_global();
 struct mgos_imu *imu = mgos_imu_create();
 struct mgos_imu_acc_opts acc_opts;
 struct mgos_imu_gyro_opts gyro_opts;
 struct mgos_imu_mag_opts mag_opts;

 if (!i2c) {
 LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
 return false;
 }

 if (!imu) {
 LOG(LL_ERROR, ("Cannot create IMU"));
 return false;
 }

 acc_opts.type = ACC_MPU9250;
 acc_opts.scale = 16.0; // G
 acc_opts.odr = 100; // Hz
 if (!mgos_imu_accelerometer_create_i2c(imu, i2c, 0x68, &acc_opts))
 LOG(LL_ERROR, ("Cannot create accelerometer on IMU"));

 gyro_opts.type = GYRO_MPU9250;
 gyro_opts.scale = 2000; // deg/sec
 gyro_opts.odr = 100; // Hz
 if (!mgos_imu_gyroscope_create_i2c(imu, i2c, 0x68, &gyro_opts))
 LOG(LL_ERROR, ("Cannot create gyroscope on IMU"));

 mag_opts.type = MAG_AK8963;
 mag_opts.scale = 12.0; // Gauss
 mag_opts.odr = 10; // Hz
 if (!mgos_imu_magnetometer_create_i2c(imu, i2c, 0x0C, &mag_opts))
 LOG(LL_ERROR, ("Cannot create magnetometer on IMU"));

 mgos_set_timer(1000, true, imu_cb, imu);
 return true;
}

```

# Demo Code

For a cool demo, take a look at my [Demo Apps](#):

## Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

## SSD1306 OLED Display Driver Support

This library adds support for the Solomon Systech SSD1306 OLED display driver to Mongoose OS.

The default configuration and PIN mapping supports ESP32 boards with the driver hard-wired for I2C communication via GPIO pins 4 and 5. It should work on other platforms, but has not yet been tested.

This driver should support displays of any resolution supported by the SSD1306.

This driver does not yet support SPI operation.

## Note about the [LOLIN / WEMOS OLED shield](#)

It uses a 64x48 panel with column offset of 32, correct configuration for it is as follows:

```
config_schema:
 - ["i2c.enable", true]
 - ["i2c.sda_gpio", 4]
 - ["i2c.scl_gpio", 5]
 - ["ssd1306.i2c.enable", false] # Use system bus.
 - ["ssd1306.enable", true]
 - ["ssd1306.width", 64]
 - ["ssd1306.height", 48]
 - ["ssd1306.col_offset", 32]
```

## Implementation of Mongoose OS RPC over UART

## Low level Bluetooth support

Currently contains only GATT server implementation for ESP32.

## Configuration section

`bt-common` library adds a `bt` configuration section with the following settings:

```

"bt": {
 "enable": true, // Enabled by default. Disabled on first reboot with WiFi on
 "dev_name": "", // Device name. If empty, value equals to device.id
 "adv_enable": true, // Advertise our Bluetooth services
 "keep_enabled": true, // Keep enabled after successful boot with WiFi on
 "scan_rsp_data_hex": "", // Custom scan response data, as hex string (e.g. `48656c6c6f` for `Hello`)
 "allow_pairing": true, // Allow pairing/bonding with other devices
 "max_paired_devices": 10, // Allow pairing with up to this many devices; -1 - no limit
 "gatts": {
 "min_sec_level": 0, // Minimum security level for all attributes of all services.
 // 0 - no auth required, 1 - encryption reqd, 2 - encryption + MITM reqd
 "require_pairing": false // Require that device is paired before accessing services
 }
}

```

## Security

Default settings allow for unrestricted access: anyone can pair with a device and access the services.

A better idea is to set `bt.gatts.require_pairing` to true, `bt.allow_pairing` to false and only enable it for a limited time via `mgos_bt_gap_set_pairing_enable` when user performs some action, e.g. presses a button.

Raising `bt.gatts.min_sec_level` to at least 1 is also advisable.

*Note:* At present, level 2 (MITM protection) is not usable as it requires device to have at least output capability during pairing, and there's no API for displaying the pairing code yet.

## A DHT Mongoose OS library

### Overview

This is a library for the DHT series of low cost temperature/humidity sensors.  
# Library to read Dallas temperature sensor using ESP32's RMT device

The library uses the `onewire-rmt` library.

A pullup resistor (4.7k) must be connected between the data line of the sensor and Vcc.

### Example code in C++

```

#include <mgos.h>
<p class="mume-header " id="include-mgosh-31"></p>

#include "DallasRmt.h"
<p class="mume-header " id="include-dallasrmth"></p>

static DallasRmt* dallas = NULL;
static OneWireRmt* ow = NULL;
static int numDevices = 0;
static uint32_t readings = 0;

static void loop_onewire(void *arg)
{
 (void) arg;
 if (0 == numDevices) {
 LOG(LL_WARN, ("No device found"));
 return;
 }
 bool wait = dallas->getWaitForConversion();
 double start = mg_time();
 dallas->requestTemperatures();
 double end = mg_time();
 LOG(LL_WARN, ("wait=%d, conversionTime=%.2fms", wait, 1000.0 * (end - start)));

 uint8_t deviceAddress[8];
 for (int i = 0; i < numDevices; ++i) {
 float temp = dallas->getTempCByIndex(i);
 dallas->getAddress(deviceAddress, i);
 char buf[17];
 char*p = buf;
 for (int i = 0; i < 8; ++i) {
 p += snprintf(p, sizeof (buf) - 2 * i, "%02x", deviceAddress[i]);
 }
 LOG(LL_WARN, ("index=%d, address=%s, temp=% .2f, reading=%d", i, buf, temp, readings));
 readings++;
 }
}

extern "C"
enum mgos_app_init_result mgos_app_init(void)
{
 dallas = new DallasRmt();
 ow = new OneWireRmt(13 /*pin*/, 0 /*rmt_rx*/, 1 /*rmt_tx*/);
 dallas->setOneWire(ow);
 dallas->begin();
 numDevices = dallas->getDeviceCount();

 mgos_set_timer(3000, 1, loop_onewire, NULL);

 return MGOS_APP_INIT_SUCCESS;
}

```

## Example code in C

```

#include <mgos.h>
<p class="mume-header " id="include-mgosh-32"></p>

#include "mgos_dallas_rmt.h"
<p class="mume-header " id="include-mgos_dallas_rmth"></p>

static DallasRmt* dallas = NULL;
static int numDevices = 0;
static uint32_t readings = 0;

static void readData() {

```

```

char deviceAddress[8];
for (int i = 0; i < numDevices; ++i) {
 float temp = mgos_dallas_rmt_get_tempc_by_index(dallas, i) / 100.0;
 mgos_dallas_rmt_get_address(dallas, deviceAddress, i);
 char buf[17];
 char*p = buf;
 for (int i = 0; i < 8; ++i) {
 p += snprintf(p, sizeof (buf) - 2 * i, "%02x", deviceAddress[i]);
 }
 LOG(LL_WARN, ("index=%d, address=%s, temp=%f, reading=%d", i, buf, temp, readings));
 readings++;
}
}

static void owTimerCB(void * arg) {
 readData();
}

static void loop_onewire(void *arg) {
 (void) arg;
 if (0 == numDevices) {
 LOG(LL_WARN, ("No device found"));
 return;
 }
 bool wait = mgos_dallas_rmt_get_wait_for_conversion(dallas);
 double start = mg_time();
 mgos_dallas_rmt_request_temperatures(dallas);
 double end = mg_time();
 LOG(LL_WARN, ("wait=%d, conversionTime=%.2fms", wait, 1000.0 * (end - start)));
 if (wait) {
 readData();
 } else {
 mgos_set_timer(800, 0, owTimerCB, NULL);
 }
}

enum mgos_app_init_result mgos_app_init(void) {
 OneWireRmt* ow = mgos_onewire_rmt_create(13 /*pin*/, 0 /*rmt_rx*/, 1 /*rmt_tx*/);
 dallas = mgos_dallas_rmt_create(ow);

 mgos_dallas_rmt_begin(dallas);
 numDevices = mgos_dallas_rmt_get_device_count(dallas);

 int resolution = mgos_dallas_rmt_get_global_resolution(dallas);
 int16_t millis = mgos_dallas_rmt_millis_to_wait_for_conversion(dallas, resolution);
 LOG(LL_WARN, ("resolution=%d, millis=%d", resolution, millis));
 //mgos_dallas_rmt_set_wait_for_conversion(dallas, false);

 mgos_set_timer(3000, 1, loop_onewire, NULL);

 return MGOS_APP_INIT_SUCCESS;
}

WiFi support
<p class="mume-header " id="wifi-support"></p>
```

This library provides common WiFi API.

```
Configuration
<p class="mume-header " id="configuration-5"></p>
```

A library adds a `wifi` configuration entry with two sub-entries: `wifi.ap` which contains configuration settings for the Access Point mode, and `wifi.sta` for the Station mode.

```
Station configuration
<p class="mume-header " id="station-configuration"></p>
```

```

```javascript
"wifi": {
  "sta": {
    "enable": true,      // Enable Station mode
    "ssid": "",         // WiFi network name
    "pass": "",          // Password
    "user": "",          // Username for WPA-PEAP mode
    "anon_identity": "", // Anonymous identity for WPA mode
    "cert": "",          // Client certificate for WPA-TTLS mode
    "key": "",           // Client key for WPA-TTLS mode
    "ca_cert": "",       // CA certificate for WPA-enterprise mode
    "ip": "",            // Static IP Address
    "netmask": "",        // Static Netmask
    "gw": "",             // Static Default Gateway
    "nameserver": "",     // DNS Server
    "dhcp_hostname": ""   // Host name to include in DHCP requests
  },
  "sta1": {
    ...
  },
  "sta2": {
    ...
  },
  "sta_cfg_idx": 0,      // Station config index to start connecting with, 0, 1 or 2.
  "sta_connect_timeout": 30 // Timeout for connection, seconds.
}
```

```

## Multiple Station Configurations

Station configurations will be tried starting from `sta_cfg_idx` and each one that is enabled will be given `sta_connect_timeout` seconds to connect. Successfully connected station's index will be saved in `sta_cfg_idx` so next boot will start with previously used configuration.

Setting `sta_connect_timeout` to 0 disables this logic.

## Access Point configuration

```

"wifi": {
 "ap": {
 "enable": true, // Enable Access Point mode
 "ssid": "Mongoose_??????", // SSID to use. ?? symbols are substituted by MAC address
 "pass": "Mongoose", // Password
 "hidden": false, // Hide WiFi network
 "channel": 6, // WiFi channel
 "max_connections": 10, // Maximum number of connections
 "ip": "192.168.4.1", // Static IP Address
 "netmask": "255.255.255.0", // Static Netmask
 "gw": "192.168.4.1", // Static Default Gateway
 "dhcp_start": "192.168.4.2", // DHCP Start Address
 "dhcp_end": "192.168.4.100", // DHCP End Address
 "trigger_on_gpio": -1 // Trigger AP on low GPIO
 }
}
```

```

IBM Watson IoT Platform support library for Mongoose OS

See [IBM Watson tutorial](#)

Arduino Wire (I2C) library for Mongoose OS

This library provides an Arduino compatibility layer for the I2C protocol by

providing a `Wire.h` public header, so that one could pick an existing Arduino program which uses I2C, throw it into the Mongoose OS application sources, and ideally, it "just works".

This library provides SPI ([Serial Peripheral Interface](#)) support for Mongoose OS.

Mongoose OS Driver for TI BQ27421 Fuel Gauge

Overview

This library contains driver for the [TI BQ27421](#) li-ion battery fuel gauge.

Example usage

Initialize and read some registers:

```
#include "mgos.h"
#include "mgos_bq27421.h"

struct mgos_bq27421 *g = mgos_bq27421_create(mgos_i2c_get_global());

uint16_t dt = 0, fwv = 0, dcap = 0;

if (!mgos_bq27421_ctl_read(g, MGOS_BQ27421_CTL_DEVICE_TYPE, &dt) ||
    !mgos_bq27421_ctl_read(g, MGOS_BQ27421_CTL_FW_VERSION, &fwv) ||
    !mgos_bq27421_reg_read(g, MGOS_BQ27421_REG_CAP_DESIGN, &dcap)) {
    // Error
}

LOG(LL_INFO, ("Gauge type 0x%x, fw version 0x%x, battery design capacity %u mAh", dt, fwv, dcap));
```

Update battery design capacity (signed 16-bit integer in data class 82 at offset 10):

```
if (!mgos_bq27421_data_write_i2(g, 82, 10, 1234 /* mAh */)) {
    // Error
}
```

The rest of the API is documented in the [include/mgos_bq27421.h](header file).

Acknowledgements

This driver is contributed by [Watchdog System](#).

License

Released under Apache License 2.0, see [LICENSE](#) file.

Implementation of Mongoose OS RPC over MQTT protocol

Overview

MQTT RPC channel allows invoking RPC calls via MQTT.

Topics

When `rpc.mqtt.enable` is true (default), device subscribes to `rpc.mqtt.sub_topic` (default: `${device.id}/rpc`, e.g. `esp8266_DA7E15/rpc` in the example).

Responses are published to `${src}/rpc`, where `${src}` is taken from the request's frame, so requests for which a response is expected must include `src`.

Examples

Request

Published to `esp8266_DA7E15/rpc`:

```
{ "id": 123, "src": "foo", "method": "Sys.GetInfo"}
```

Response will be published to `foo/rpc`.

mos tool support

`mos` tool supports RPC over MQTT via the `mqtt` port schema:

```
$ mos --port mqtt://iot.eclipse.org/esp8266_DA7E15 call Sys.GetInfo
```

For brokers that require TLS, `mqtts` should be used:

```
$ mos --port mqtts://iot.eclipse.org/esp8266_DA7E15 call Sys.GetInfo
```

For brokers that require TLS client authentication (e.g. AWS), cert and file should be supplied:

```
$ mos --port mqtts://XXXXXXXXX.iot.REGION.amazonaws.com/esp8266_DA7E15 --cert-file mycert.pem --key-file mykey.pem call Sys.GetInfo
```

Bluetooth Current Time Service implementation

Implements the [Current Time GATT service](#).

Note: partial implementation, no notifications or time reference info.

Copyright (c) 2019 Deomid "rojer" Ryabkov

All rights reserved

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

```
http://www.apache.org/licenses/LICENSE-2.0
```

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Overview

Usage

```
#include "mgos_lolin_button.h"

static void lolin_button_handler(int ev, void *ev_data, void *userdata) {
    const struct mgos_lolin_button_status *bs = (const struct mgos_lolin_button_status *) ev_data;
    const char *bn = NULL;
    switch (ev) {
        case MGOS_EV_LOLIN_BUTTON_A:
            bn = "A";
            break;
        case MGOS_EV_LOLIN_BUTTON_B:
            bn = "B";
            break;
        default:
            return;
    }
    const char *sn = NULL;
    switch (bs->state) {
        case MGOS_LOLIN_BUTTON_RELEASE:
            sn = "released";
            break;
        case MGOS_LOLIN_BUTTON_PRESS:
            sn = "pressed";
            break;
        case MGOS_LOLIN_BUTTON_DOUBLE_PRESS:
            sn = "double-pressed";
            break;
        case MGOS_LOLIN_BUTTON_LONG_PRESS:
            sn = "long-pressed";
            break;
        case MGOS_LOLIN_BUTTON_HOLD:
            sn = "held";
            break;
    }
    LOG(LL_INFO, ("Button %s %s", bn, sn));
    (void) userdata;
}

mgos_event_add_group_handler(MGOS_EV_LOLIN_BUTTON_BASE, lolin_button_handler, NULL);
```

BH1750 ambient light sensor native library for Mongoose OS

Overview

BH1750 ambient light sensor native C driver for Mongoose OS

Sensor wiring

Todo

- implement sensor sensitivity adjusting
- use timer instead of mgos_msleep?

Usage

```
struct sensor_bh1750* bh = sensor_bh1750_create(0x23);

float lux = sensor_bh1750_get_lux(bh, ONE_TIME_HIGH_RES_MODE_2);
printf("Light: %f lux\n", lux);

sensor_bh1750_free(bh, false);
```# A native Mongoose OS DS3231 RTC library

Implementation
<p class="mume-header" id="implementation"></p>

The library implements 2 structures: `struct mgos_ds3231` and a helper `struct mgos_ds3231_date_time`. Both structures are available for mJS too. The mJS support is available if the user includes the mjs library:
```

- origin: <https://github.com/mongoose-os-libs/mjs>

```
struct mgos_ds3231
<p class="mume-header" id="struct-mgos_ds3231"></p>
```

This is used to communicate with the DS3231.  
The RTC data is read into and written from using the helper `struct mgos\_ds3231\_date\_time`

```
Example in C
<p class="mume-header" id="example-in-c"></p>
```

```
```c
struct mgos_ds3231* ds=mgos_ds3231_create(addr);
time_t unixtime=time(NULL);
mgos_ds3231_write_unixtime(ds, unixtime);
mgos_ds3231_free(ds);
```

Example in mJS

```
let ds=DS3231.create(addr);
let unixtime=Timer.now();
ds.writeUnixtime(unixtime);
ds.free();
```

struct mgos_ds3231_date_time

Encapsulates the DS3231 date/time information plus the unix timestamp.
Several functions are provided to create and free the structure and to get/set different fields.

Example in C

```
struct mgos_ds3231_date_time* dt=mgos_ds3231_date_time_create();
mgos_ds3231_date_time_set_date(dt, 2016, 2, 3);
// mgos_ds3231_date_time_set_time will calculate the unixtime.
// It is VERY important to call mgos_ds3231_date_time_set_date first
mgos_ds3231_date_time_set_time(dt, 12, 34, 56);

struct mgos_ds3231* ds=mgos_ds3231_create(addr);
mgos_ds3231_write(ds, dt);
mgos_ds3231_settimeofday(ds);

mgos_ds3231_free(ds);
mgos_ds3231_date_time_free(dt);
```

Example in mJS

```
let dt=DS3231DateTime.create();
dt.setDate(2016, 2, 3);
dt.setTime(12, 34, 56);

let ds=DS3231.create(addr);
ds.write(dt);
ds.setTimeOfDay();

ds.free();
dt.free();
```

Winbond W25XXX SPI NAND Flash Driver

This is a VFS device driver for [Winbond W25 series](#) SPI NAND flash chips.
W25N01 and W25M02 (2Gb multi-die version) are supported.

BH1730 sensor support

This library provides support for the digital 16-bit ambient light sensor BH1730.

See API documentation: for [C](#)
and [mJS](#).

RPC - Remote Procedure Calls

See [MG-RPC Guide](#)
for detailed documentation.

A Cron Mongoose OS library

Overview

This library implement cron core functionality. See also:
[crontab](#).

This library supports classic cron expressions with additional field for the seconds, plus a few extras. We'll get back to the extras later, let's start with regular cron expressions.

Classic cron expressions.

Each expression contains 6 fields:

- seconds, 0-59
- minutes, 0-59
- hours, 0-23
- day of month, 1-31
- month, 1-12 or JAN-DEC
- day of week 0-6 or SUN-SAT

Each field can contain multiple values separated with commas, and/or ranges

determined by the beginning of the range, a hyphen, and the ending of the range. For example, for the day of week, it could be MON-THU,SAT

A slash can be used to specify intervals: e.g. */5 in seconds field means "every 5 seconds".

Each field can contain an asterisk * which means "any value".

Examples:

- */15 * 1-4 * * * : Run every 15 seconds from 1 to 4 hours;
- 0 */2 1-4 * * * : Run every two minutes from 1 to 4 hours;
- 0 0 7 * * MON-FRI : Run at 7:00 every working day;
- 0 30 23 30 * * : Run at 23:30 every 30th day of month.

Randomized cron expressions

It's possible to specify the time range in which a job will run certain number of times.

For example, this:

```
@random:{"from":"10 * * * *", "to":"50 * * * *", "number":5}
```

means that the cron job will be fired in between of 10 and 50 seconds of every minute, and the total number of invocations in that 40-second time window will be approximately 5. It's not guaranteed to be exactly 5 though: might be a bit less or more.

Another example: approximately 10 invocations in between of 08:00 and 22:00 on working days:

```
@random:{"from":"0 0 8 * * MON-FRI", "to":"0 0 22 * * MON-FRI", "number":10}
```

Note that the timeout between from and to should always be the same.

For example, this would make no sense:

```
@random:{"from":"0 0 8 * * MON-FRI", "to":"0 0 22 * * *", "number":10}
```

Currently these cases are not checked yet, so the above expression will just result in incorrect behavior.

Sunrise/sunset

Finally, instead of specifying exact time with the first three fields (seconds, minutes, hours) one could use @sunrise / @sunset, plus optional offset like +2h45m or -1.5h.

For that to work, the device needs to have latitude/longitude and timezone configured.

Examples:

- @sunrise * * MON-FRI : Run at the sunrise on working days;
- @sunset-1h30m 1 * * : Run 1.5 hours before the sunset every 1th day of month;
- @sunset * * * : Run at the sunset every day

As a shortcut for the last case, when all fields are *, these fields can be omitted. Thus, more valid examples:

- @sunrise
- @sunset
- @sunset+3h

A collection of libraries for demoing Mongoose OS

Overview

This library is intended to be used for apps that demonstrate Mongoose OS capabilities. It is essentially a collection of libraries that implement a wide set of functionalities - from hardware peripherals API to cloud integrations like AWS IoT, Google IoT Core, etc.

Storage device and filesystem init table

Overview

Configures storage devices, creates and/or mounts filesystems according to configuration.

Configuration

This library defines two configuration sections: `devtab` (5 entries) and `fstab` (3 entries).

`Devtab` defines VFS devices to be created, `fstab` defines filesystems to create and/or mount.

`Devtab` is processed first, then `fstab`.

See `config_schema` section of the [manifest](#) for detailed description of settings.

Examples

- External SPI flash formatted as LFS and mounted on `/data`. Size is auto-detected.

```
libs:
  - origin: https://github.com/mongoose-os-libs/fstab
  - origin: https://github.com/mongoose-os-libs/vfs-dev-spi-flash
  - origin: https://github.com/mongoose-os-libs/vfs-fs-lfs

config_schema:
  - ["spi.enable", true]
  # Other SPI interface options go here.
  - ["devtab.dev0.name", "spif0"]
  - ["devtab.dev0.type", "spi_flash"]
  - ["devtab.dev0.opts", '{"cs": 0, "freq": 10000000}']
  - ["fstab.fs0.dev", "spif0"]
  - ["fstab.fs0.type", "LFS"]
  - ["fstab.fs0.opts", {"bs": 4096}]
  - ["fstab.fs0.path", "/data"]
  - ["fstab.fs0.create", true]
```

- (ESP32) Additional partition on the ESP32 system flash, formatted as LFS and mounted on `/data`.

Note: All the data ESP32 partitions are automatically registered so there are no explicit devtab entries.

```

libs:
- origin: https://github.com/mongoose-os-libs/fstab
- origin: https://github.com/mongoose-os-libs/vfs-fs-lfs

build_vars:
ESP_IDF_EXTRA_PARTITION: data,data,spiffs,,256K

config_schema:
- ["fstab.fs0.dev", "data"]
- ["fstab.fs0.type", "LFS"]
- ["fstab.fs0.opts", '{"bs": 4096}']
- ["fstab.fs0.path", "/data"]
- ["fstab.fs0.create", true]

```

- External SPI flash split into two parts

First formatted for SPIFFS, the rest is not used.

```

libs:
- origin: https://github.com/mongoose-os-libs/fstab
- origin: https://github.com/mongoose-os-libs/vfs-dev-part
- origin: https://github.com/mongoose-os-libs/vfs-dev-spi-flash
- origin: https://github.com/mongoose-os-libs/vfs-fs-spiffs

config_schema:
- ["spi.enable", true]
# Other SPI interface options go here.
- ["devtab.dev0.name", "spif0"]
- ["devtab.dev0.type", "spi_flash"]
- ["devtab.dev0.opts", '{"cs": 0, "freq": 10000000}']
- ["devtab.dev1.name", "spif0p1"]
- ["devtab.dev1.type", "part"]
- ["devtab.dev1.opts", {"dev": "spif0", "offset": 0, "size": 131072}]
- ["devtab.dev2.name", "spif0p2"]
- ["devtab.dev2.type", "part"]
- ["devtab.dev2.opts", {"dev": "spif0", "offset": 131072}]
- ["fstab.fs0.dev", "spif0p1"]
- ["fstab.fs0.type", "SPIFFS"]
- ["fstab.fs0.opts", {"bs": 4096, "ps": 128, "es": 4096}]
- ["fstab.fs0.path", "/data"]
- ["fstab.fs0.create", true]

```

VFS for RAM

Location library

Overview

Provides a function to get latitude and longitude; so far it merely returns the configured values. Example usage, in the app's `mos.yml`:

```

libs:
- origin: https://github.com/mongoose-os-libs/location

config_schema:
- ["device.location.lat", 53.3242381]
- ["device.location.lon", -6.385785]

```

A blank Mongoose OS library

Overview

This blank library is intended to be used as a skeleton for developing Mongoose OS libraries.[# How to contribute](#)

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

INA219 I2C Driver

A Mongoose library for Texas Instruments' popular I2C current sensor.

Implementation details

The INA219 is a simple I2C device that measures a voltage drop over a shunt resistor, as well as the voltage on the bus. It is able to measure voltage up to 26V (in 4mV increments) on the bus and the voltage drop over the shunt resistor in 10uV increments with a range of 320mV, and is reasonably precise.

Using a 0.1Ohm shunt resistor, the maximum current that can be measured is 3.2A.

API Description

Create an INA219 object using `mgos_ina219_create()`, set the shunt resistor value using `mgos_ina219_set_shunt_resistance()`, then call `mgos_ina219_get_bus_voltage()` to get the bus voltage, call `mgos_ina219_get_shunt_voltage()` to get the voltage drop over the shunt resistor, and `mgos_ina219_get_current()` to get the current flowing through the circuit.

Example application

```
#include "mgos.h"
#include "mgos_config.h"
#include "mgos_ina219.h"

static void ina219_timer_cb(void *user_data) {
    struct mgos_ina219 *sensor = (struct mgos_ina219 *)user_data;
    float bus, shunt, current, res;

    if (!sensor) return;

    mgos_ina219_get_bus_voltage(sensor, &bus);
    mgos_ina219_get_shunt_resistance(sensor, &res);
    mgos_ina219_get_shunt_voltage(sensor, &shunt);
    mgos_ina219_get_current(sensor, &current);
    LOG(LL_INFO, ("Vbus=% .3f V Vshunt=% .0f uV Rshunt=% .3f Ohm Ishunt=% .1f mA",
        bus, shunt*1e6, res, current*1e3));
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_ina219 *sensor;

    sensor = mgos_ina219_create(mgos_i2c_get_global(), mgos_sys_config_get_ina219_i2caddr());
    if (!sensor) {
        LOG(LL_ERROR, ("Could not create INA219 sensor"));
        return false;
    }

    mgos_set_timer(1000, true, ina219_timer_cb, sensor);

    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

RPC Service - Filesystem

This service provides an ability to manage device files remotely.

It is required by the `mos ls`, `mos get`, `mos put`, `mos rm` commands.

If this library is not included in the app, those commands won't work.

It is possible to call this service programmatically via serial, HTTP/RESTful,

Websocket, MQTT or other transports

(see [RPC section](#)) or use `mos` tool.



Below is a list of exported RPC methods and arguments:

FS.List

Get device file list. Arguments: none.

Example usage:

```
mos call FS.List
[
  "mgos_ro_vars_schema.json",
  "conf0.json",
  ...
]
```

This RPC command has a shortcut: `mos ls`:

```
mos ls
"mgos_ro_vars_schema.json",
"conf0.json",
...
```

FS.ListExt

Same as `FS.List` but also returns extra file info like file sizes.

Arguments: none.

Example usage:

```
mos call FS.ListExt
[
{
  "name": "mgos_ro_vars_schema.json",
  "size": 332
},
...
]
```

This RPC command has a shortcut: `mos ls -l`:

```
mos ls -l
api_adc.js 259
api_arch_uart.js 651
...
```

FS.Get

Returns file content. Arguments:

```
{
  "filename": "foo.txt",      // Required. Name of the file to fetch.
  "offset": 0,                // Optional. Offset to begin with.
  "len": 100                 // Optional. Number of bytes to return.
}
```

Example usage:

```
mos call FS.Get '{"filename": "init.js"}'
{
  "data": "bG9hZCgnYXBpX2NvbmlZpZ...",    # Base64 encoded data
  "left": 0                                # How many bytes left in a file
}
```

This RPC command has a shortcut: `mos get`:

```
mos get init.js
load('api_gpio.js');
...
```

FS.Put

Write data into file. Write is done either by overwriting an existing content, or by appending to the existing content. Arguments:

```
{
  "filename": "foo.txt",      // Required. Name of the file to write to.
  "append": false,            // Optional. Overwrite or append.
  "data": "base64 text"      // Required. Data to write.
}
```

This RPC command has a shortcut: `mos put`. It splits large files into small chunks, and calls `FS.Put` sequentially, appending content.

```
mos put /etc/passwd foo.txt
```

FS.Remove

Delete file. Arguments:

```
{  
  "filename": "foo.txt"    // Required. Name of the file to delete  
}
```

This RPC command has a shortcut: mos rm:

```
mos rm foo.txt
```

FS.Mkfs

Create filesystem. Arguments:

```
{  
  "dev_type": "spi_flash",      // Required. Filesystem driver name.  
  "dev_opts": "...",          // Required. Device-specific options.  
  "fs_type": "SPIFFS",         // Required. Filesystem type.  
  "fs_opts": "{\"site\": 131072}" // Required. Filesystem-specific options.  
}
```

Example usage:

```
mos call FS.Mkfs '{"dev_type": "spi_flash", "dev_opts": "{\"freq\": 20000000, \"cs\": 0}", "fs_type": "SPIFFS", "fs_opts": {}}
```

FS.Mount

Mount filesystem. Arguments:

```
{  
  "path": "/foo",           // Required. Directory name to attach to.  
  "dev_type": "...",       // See FS.Mkfs above  
  "dev_opts": "...",  
  "fs_type": "SPIFFS",  
  "fs_opts": "..."  
}
```

```
mos call FS.Mount '{"path": "/mnt", "dev_type": "spi_flash", "dev_opts": "{\"freq\": 20000000, \"cs\": 0}", "fs_type": "SPIFFS", "fs_opts": {}}
```

FS.Umount

Unmout filesystem. Arguments:

```
{  
  "path": "/foo"           // Required. Mount point to detach.  
}
```

```
mos call FS.Umount '{"path": "/mnt"}'
```

A VFS device that exposes a part of another device

Example

```
vfs_dev_create("big0part", "{\"dev\": \"big0\", \"offset\": 32768, \"size\": 65536}");
```

Onewire library for Mongoose OS using the ESP32's RMT device

Based on <https://github.com/nodemcu/nodemcu-firmware/blob/dev-esp32/components/platform/onewire.c>

It uses 2 RMT channels, one for RX and one for TX.

Create the onewire object like this:

```
OnewireRmt* mgos_onewire_rmt_create(int pin,int rmt_rx,int rmt_tx);
```

or

```
OnewireRmt(uint8_t pin,uint8_t rmt_rx,uint8_t rmt_tx);
```

The design of the little filesystem

A little fail-safe filesystem designed for embedded systems.



For a bit of backstory, the littlefs was developed with the goal of learning more about filesystem design by tackling the relative unsolved problem of managing a robust filesystem resilient to power loss on devices with limited RAM and ROM.

The embedded systems the littlefs is targeting are usually 32 bit microcontrollers with around 32KB of RAM and 512KB of ROM. These are often paired with SPI NOR flash chips with about 4MB of flash storage.

Flash itself is a very interesting piece of technology with quite a bit of nuance. Unlike most other forms of storage, writing to flash requires two operations: erasing and programming. The programming operation is relatively cheap, and can be very granular. For NOR flash specifically, byte-level programs are quite common. Erasing, however, requires an expensive operation that forces the state of large blocks of memory to reset in a destructive reaction that gives flash its name. The [Wikipedia entry](#) has more information if you are interested in how this works.

This leaves us with an interesting set of limitations that can be simplified to three strong requirements:

1. **Power-loss resilient** - This is the main goal of the littlefs and the focus of this project.
Embedded systems are usually designed without a shutdown routine and a notable lack of user interface for recovery, so filesystems targeting

embedded systems must be prepared to lose power at any given time. Despite this state of things, there are very few embedded filesystems that handle power loss in a reasonable manner, and most can become corrupted if the user is unlucky enough.

2. **Wear leveling** - Due to the destructive nature of flash, most flash chips have a limited number of erase cycles, usually in the order of around 100,000 erases per block for NOR flash. Filesystems that don't take wear into account can easily burn through blocks used to store frequently updated metadata.

Consider the [FAT filesystem](#), which stores a file allocation table (FAT) at a specific offset from the beginning of disk. Every block allocation will update this table, and after 100,000 updates, the block will likely go bad, rendering the filesystem unusable even if there are many more erase cycles available on the storage as a whole.

3. **Bounded RAM/ROM** - Even with the design difficulties presented by the previous two limitations, we have already seen several flash filesystems developed on PCs that handle power loss just fine, such as the logging filesystems. However, these filesystems take advantage of the relatively cheap access to RAM, and use some rather... opportunistic... techniques, such as reconstructing the entire directory structure in RAM. These operations make perfect sense when the filesystem's only concern is erase cycles, but the idea is a bit silly on embedded systems.
To cater to embedded systems, the littlefs has the simple limitation of using only a bounded amount of RAM and ROM. That is, no matter what is written to the filesystem, and no matter how large the underlying storage is, the littlefs will always use the same amount of RAM and ROM. This presents a very unique challenge, and makes presumably simple operations, such as iterating through the directory tree, surprisingly difficult.

Existing designs?

There are of course, many different existing filesystem. Here is a very rough summary of the general ideas behind some of them.

Most of the existing filesystems fall into the one big category of filesystem designed in the early days of spinny magnet disks. While there is a vast amount of interesting technology and ideas in this area, the nature of spinny magnet disks encourage properties, such as grouping writes near each other, that don't make as much sense on recent storage types. For instance, on flash, write locality is not important and can actually increase wear.

One of the most popular designs for flash filesystems is called the

[logging filesystem](#).

The flash filesystems [jffs](#) and [yafts](#) are good examples. In a logging filesystem, data is not stored in a data structure on disk, but instead the changes to the files are stored on disk. This has several neat advantages, such as the fact that the data is written in a cyclic log format and naturally wear levels as a side effect. And, with a bit of error detection, the entire filesystem can easily be designed to be resilient to power loss. The journaling component of most modern day filesystems is actually a reduced form of a logging filesystem. However, logging filesystems have a difficulty scaling as the size of storage increases. And most filesystems compensate by caching large parts of the filesystem in RAM, a strategy that is inappropriate

for embedded systems.

Another interesting filesystem design technique is that of [copy-on-write \(COW\)](#).

A good example of this is the [btrfs](#)

filesystem. COW filesystems can easily recover from corrupted blocks and have natural protection against power loss. However, if they are not designed with wear in mind, a COW filesystem could unintentionally wear down the root block where the COW data structures are synchronized.

Metadata pairs

The core piece of technology that provides the backbone for the littlefs is the concept of metadata pairs. The key idea here is that any metadata that needs to be updated atomically is stored on a pair of blocks tagged with a revision count and checksum. Every update alternates between these two pairs, so that at any time there is always a backup containing the previous state of the metadata.

Consider a small example where each metadata pair has a revision count, a number as data, and the XOR of the block as a quick checksum. If we update the data to a value of 9, and then to a value of 5, here is what the pair of blocks may look like after each update:

```
block 1  block 2      block 1  block 2      block 1  block 2
-----|-----|-----|-----|-----|-----|
| rev: 1 | rev: 0 | | rev: 1 | rev: 2 | | rev: 3 | rev: 2 |
| data: 3 | data: 0 | -> | data: 3 | data: 9 | -> | data: 5 | data: 9 |
| xor: 2 | xor: 0 | | xor: 2 | xor: 11 | | xor: 6 | xor: 11 |
-----|-----|-----|-----|-----|
          let data = 9          let data = 5
```

After each update, we can find the most up to date value of data by looking at the revision count.

Now consider what the blocks may look like if we suddenly lose power while changing the value of data to 5:

```
block 1  block 2      block 1  block 2      block 1  block 2
-----|-----|-----|-----|-----|-----|
| rev: 1 | rev: 0 | | rev: 1 | rev: 2 | | rev: 3 | rev: 2 |
| data: 3 | data: 0 | -> | data: 3 | data: 9 | -x | data: 3 | data: 9 |
| xor: 2 | xor: 0 | | xor: 2 | xor: 11 | | xor: 2 | xor: 11 |
-----|-----|-----|-----|-----|
          let data = 9          let data = 5
                           powerloss!!!
```

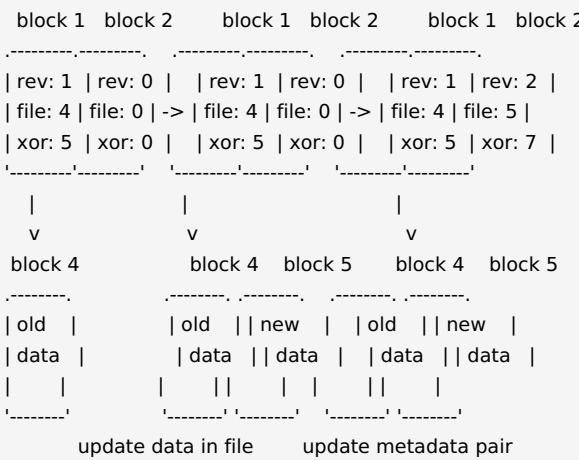
In this case, block 1 was partially written with a new revision count, but the littlefs hadn't made it to updating the value of data. However, if we check our checksum we notice that block 1 was corrupted. So we fall back to block 2 and use the value 9.

Using this concept, the littlefs is able to update metadata blocks atomically. There are a few other tweaks, such as using a 32 bit CRC and using sequence arithmetic to handle revision count overflow, but the basic concept is the same. These metadata pairs define the backbone of the littlefs, and the rest of the filesystem is built on top of these atomic updates.

Non-meta data

Now, the metadata pairs do come with some drawbacks. Most notably, each pair requires two blocks for each block of data. I'm sure users would be very unhappy if their storage was suddenly cut in half! Instead of storing everything in these metadata blocks, the littlefs uses a COW data structure for files which is in turn pointed to by a metadata block. When we update a file, we create copies of any blocks that are modified until the metadata blocks are updated with the new copy. Once the metadata block points to the new copy, we deallocate the old blocks that are no longer in use.

Here is what updating a one-block file may look like:



It doesn't matter if we lose power while writing new data to block 5, since the old data remains unmodified in block 4. This example also highlights how the atomic updates of the metadata blocks provide a synchronization barrier for the rest of the littlefs.

At this point, it may look like we are wasting an awfully large amount of space on the metadata. Just looking at that example, we are using three blocks to represent a file that fits comfortably in one! So instead of giving each file a metadata pair, we actually store the metadata for all files contained in a single directory in a single metadata block.

Now we could just leave files here, copying the entire file on write provides the synchronization without the duplicated memory requirements of the metadata blocks. However, we can do a bit better.

CTZ skip-lists

There are many different data structures for representing the actual files in filesystems. Of these, the littlefs uses a rather unique [COW](#) data structure that allows the filesystem to reuse unmodified parts of the file without additional metadata pairs.

First lets consider storing files in a simple linked-list. What happens when we append a block? We have to change the last block in the linked-list to point to this new block, which means we have to copy out the last block, and change the second-to-last block, and then the third-to-last, and so on until we've copied out the entire file.

Exhibit A: A linked-list

```
-----  
| data 0 |->| data 1 |->| data 2 |->| data 4 |->| data 5 |->| data 6 |  
|   | |   | |   | |   | |   | |  
'-----' '-----' '-----' '-----'
```

To get around this, the littlefs, at its heart, stores files backwards. Each block points to its predecessor, with the first block containing no pointers. If you think about for a while, it starts to make a bit of sense. Appending blocks just point to their predecessor and no other blocks need to be updated. If we update a block in the middle, we will need to copy out the blocks that follow, but can reuse the blocks before the modified block. Since most file operations either reset the file each write or append to files, this design avoids copying the file in the most common cases.

Exhibit B: A backwards linked-list

```
-----  
| data 0 |<-| data 1 |<-| data 2 |<-| data 4 |<-| data 5 |<-| data 6 |  
|   | |   | |   | |   | |  
'-----' '-----' '-----'
```

However, a backwards linked-list does come with a rather glaring problem. Iterating over a file *in order* has a runtime cost of $O(n^2)$. Gah! A quadratic runtime to just *read* a file? That's awful. Keep in mind reading files is usually the most common filesystem operation.

To avoid this problem, the littlefs uses a multilayered linked-list. For every n th block where n is divisible by 2^x , the block contains a pointer to block $n-2^x$. So each block contains anywhere from 1 to $\log_2(n)$ pointers that skip to various sections of the preceding list. If you're familiar with data-structures, you may have recognized that this is a type of deterministic skip-list.

The name comes from the use of the [count trailing zeros \(CTZ\)](#) instruction, which allows us to calculate the power-of-two factors efficiently. For a given block n , the block contains $\text{ctz}(n)+1$ pointers.

Exhibit C: A backwards CTZ skip-list

```
-----  
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 |<-| data 4 |<-| data 5 | |
|   |<-|   |-|   |<-|   |-|   | |   | |  
|   |<-|   |-|   |<-|   |-|   | |   | |  
'-----' '-----' '-----' '-----'
```

The additional pointers allow us to navigate the data-structure on disk much more efficiently than in a singly linked-list.

Taking exhibit C for example, here is the path from data block 5 to data block 1. You can see how data block 3 was completely skipped:

```
-----  
| data 0 | | data 1 |<-| data 2 | | data 3 | | data 4 |<-| data 5 |  
|   | |   | |<-|   |-|   | | | |  
|   | |   | |<-|   |-|   | | | |
```

The path to data block 0 is even more quick, requiring only two jumps:

```
graph LR; A["| data 0 |"] --> B["| data 1 |"]; B --> C["| data 2 |"]; C --> D["| data 3 |"]; D --> E["| data 4 |"]; E --> A
```

We can find the runtime complexity by looking at the path to any block from the block containing the most pointers. Every step along the path divides the search space for the block in half. This gives us a runtime of $O(\log n)$.

To get to the block with the most pointers, we can perform the same steps backwards, which puts the runtime at $O(2 \log n) = O(\log n)$. The interesting part about this data structure is that this optimal path occurs naturally if we greedily choose the pointer that covers the most distance without passing our target block.

So now we have a representation of files that can be appended trivially with a runtime of $O(1)$, and can be read with a worst case runtime of $O(n \log n)$. Given that the runtime is also divided by the amount of data we can store in a block, this is pretty reasonable.

Unfortunately, the CTZ skip-list comes with a few questions that aren't straightforward to answer. What is the overhead? How do we handle more pointers than we can store in a block? How do we store the skip-list in a directory entry?

One way to find the overhead per block is to look at the data structure as multiple layers of linked-lists. Each linked-list skips twice as many blocks as the previous linked-list. Another way of looking at it is that each linked-list uses half as much storage per block as the previous linked-list. As we approach infinity, the number of pointers per block forms a geometric series. Solving this geometric series gives us an average of only 2 pointers per block.

Finding the maximum number of pointers in a block is a bit more complicated but since our file size is limited by the integer width we use to store the size, we can solve for it. Setting the overhead of the maximum pointers equal to the block size we get the following equation. Note that a smaller block size results in more pointers, and a larger word width results in larger pointers.

where:

B = block size in bytes

w = word width in bits

Solving the equation for B gives us the minimum block size for various word widths:

32 bit CTZ skip-list = minimum block size of 104 bytes
64 bit CTZ skip-list – minimum block size of 448 bytes

Since littlefs uses a 32 bit word size, we are limited to a minimum block size of 104 bytes. This is a perfectly reasonable minimum block size, with most block sizes starting around 512 bytes. So we can avoid additional logic to avoid overflowing our block's capacity in the CTZ skip-list.

So, how do we store the skip-list in a directory entry? A naive approach would be to store a pointer to the head of the skip-list, the length of the file in bytes, the index of the head block in the skip-list, and the offset in the head block in bytes. However this is a lot of information, and we can observe

that a file size maps to only one block index + offset pair. So it should be sufficient to store only the pointer and file size.

But there is one problem, calculating the block index + offset pair from a file size doesn't have an obvious implementation.

We can start by just writing down an equation. The first idea that comes to mind is to just use a for loop to sum together blocks until we reach our file size. We can write this equation as a summation:

where:

B = block size in bytes

w = word width in bits

n = block index in skip-list

N = file size in bytes

And this works quite well, but is not trivial to calculate. This equation requires $O(n)$ to compute, which brings the entire runtime of reading a file to $O(n^2 \log n)$. Fortunately, the additional $O(n)$ does not need to touch disk, so it is not completely unreasonable. But if we could solve this equation into a form that is easily computable, we can avoid a big slowdown.

Unfortunately, the summation of the CTZ instruction presents a big challenge. How would you even begin to reason about integrating a bitwise instruction?

Fortunately, there is a powerful tool I've found useful in these situations:

The [On-Line Encyclopedia of Integer Sequences \(OEIS\)](#).

If we work out the first couple of values in our summation, we find that CTZ maps to [A001511](#), and its partial summation maps to [A005187](#), and surprisingly, both of these sequences have relatively trivial equations! This leads us to a rather unintuitive property:

where:

$\text{ctz}(x)$ = the number of trailing bits that are 0 in x

$\text{popcount}(x)$ = the number of bits that are 1 in x

It's a bit bewildering that these two seemingly unrelated bitwise instructions are related by this property. But if we start to dissect this equation we can see that it does hold. As n approaches infinity, we do end up with an average overhead of 2 pointers as we find earlier. And popcount seems to handle the error from this average as it accumulates in the CTZ skip-list.

Now we can substitute into the original equation to get a trivial equation for a file size:

Unfortunately, we're not quite done. The popcount function is non-injective, so we can only find the file size from the block index, not the other way around. However, we can solve for an n' block index that is greater than n with an error bounded by the range of the popcount function. We can then repeatedly substitute this n' into the original equation until the error is smaller than the integer division. As it turns out, we only need to perform this substitution once. Now we directly calculate our block index:

Now that we have our block index n, we can just plug it back into the above equation to find the offset. However, we do need to rearrange the equation

a bit to avoid integer overflow:

The solution involves quite a bit of math, but computers are very good at math.

Now we can solve for both the block index and offset from the file size in O(1).

Here is what it might look like to update a file stored with a CTZ skip-list:

```

        block 1  block 2
        -----.
| rev: 1 | rev: 0 |
| file: 6 | file: 0 |
| size: 4 | size: 0 |
| xor: 3 | xor: 0 |
'-----'
|
v

```

block 3 block 4 block 5 block 6

```

-----.
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 | |
|     |<-|     |--|     ||     |
|     ||     ||     ||     ||
'-----'

```

| update data in file

v

```

        block 1  block 2
        -----.
| rev: 1 | rev: 0 |
| file: 6 | file: 0 |
| size: 4 | size: 0 |
| xor: 3 | xor: 0 |
'-----'
|
v

```

block 3 block 4 block 5 block 6

```

-----.
| data 0 |<-| data 1 |<-| old  |<-| old  | |
|     |<-|     |--| data 2 | | data 3 |
|     ||     ||     ||     ||
'-----'

```

^ ^ ^

```

||     |   block 7   block 8   block 9   block 10
||     |   -----.
||     '---| new  |<-| new  |<-| new  |<-| new  |
||-----| data 2 |<-| data 3 |--| data 4 | | data 5 |
'-----|     |--|     |--|     ||     |

```

| update metadata pair

v

```

        block 1  block 2
        -----.
| rev: 1 | rev: 2 |
| file: 6 | file: 10|
| size: 4 | size: 6 |
| xor: 3 | xor: 14 |
'-----'
|

```

block 3 block 4 block 5 block 6

```

-----.
| data 0 |<-| data 1 |<-| old  |<-| old  | |
|     |<-|     |--| data 2 | | data 3 |
|     ||     ||     ||     ||
'-----'

```

^ ^ ^

```

||     |   block 7   block 8   block 9   block 10
||     |   -----.
||     '---| new  |<-| new  |<-| new  |<-| new  |
||-----| data 2 |<-| data 3 |--| data 4 | | data 5 |
'-----|     |--|     |--|     ||     |

```

Block allocation

So those two ideas provide the grounds for the filesystem. The metadata pairs give us directories, and the CTZ skip-lists give us files. But this leaves one big elephant

of a question. How do we get those blocks in the first place?

One common strategy is to store unallocated blocks in a big free list, and initially the littlefs was designed with this in mind. By storing a reference to the free list in every single metadata pair, additions to the free list could be updated atomically at the same time the replacement blocks were stored in the metadata pair. During boot, every metadata pair had to be scanned to find the most recent free list, but once the list was found the state of all free blocks becomes known.

However, this approach had several issues:

- There was a lot of nuanced logic for adding blocks to the free list without modifying the blocks, since the blocks remain active until the metadata is updated.
- The free list had to support both additions and removals in FIFO order while minimizing block erases.
- The free list had to handle the case where the file system completely ran out of blocks and may no longer be able to add blocks to the free list.
- If we used a revision count to track the most recently updated free list, metadata blocks that were left unmodified were ticking time bombs that would cause the system to go haywire if the revision count overflowed.
- Every single metadata block wasted space to store these free list references.

Actually, to simplify, this approach had one massive glaring issue: complexity.

Complexity leads to fallibility.

Fallibility leads to unmaintainability.

Unmaintainability leads to suffering.

Or at least, complexity leads to increased code size, which is a problem for embedded systems.

In the end, the littlefs adopted more of a "drop it on the floor" strategy. That is, the littlefs doesn't actually store information about which blocks are free on the storage. The littlefs already stores which files are in use, so to find a free block, the littlefs just takes all of the blocks that exist and subtract the blocks that are in use.

Of course, it's not quite that simple. Most filesystems that adopt this "drop it on the floor" strategy either rely on some properties inherent to the filesystem, such as the cyclic-buffer structure of logging filesystems, or use a bitmap or table stored in RAM to track free blocks, which scales with the size of storage and is problematic when you have limited RAM. You could iterate through every single block in storage and check it against every single block in the filesystem on every single allocation, but that would have an abhorrent runtime.

So the littlefs compromises. It doesn't store a bitmap the size of the storage, but it does store a little bit-vector that contains a fixed set lookahead for block allocations. During a block allocation, the lookahead vector is checked for any free blocks. If there are none, the lookahead region jumps forward and the entire filesystem is scanned for free blocks.

Here's what it might look like to allocate 4 blocks on a decently busy

filesystem with a 32bit lookahead and a total of
128 blocks (512Kbytes of storage if blocks are 4Kbyte):

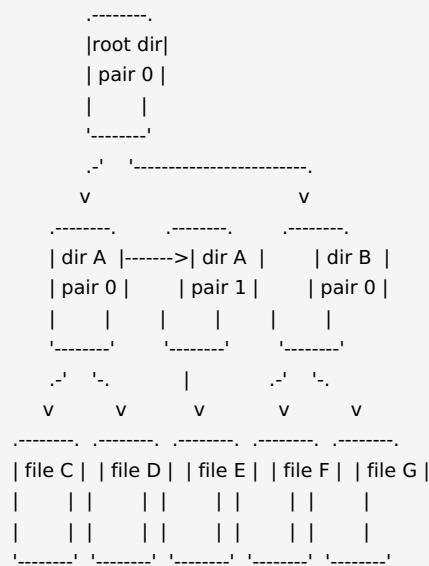
```
boot...      lookahead:  
    fs blocks: ffffff9ffffffffffffefffffff0000  
scanning...  lookahead: fffff9ff  
    fs blocks: fffff9ffffffffffffefffffff0000  
alloc = 21   lookahead: fffffdff  
    fs blocks: fffffdffffffffffffefffffff0000  
alloc = 22   lookahead: ffffffff  
    fs blocks: ffffffff0000  
scanning...  lookahead:     ffffffe  
    fs blocks: ffffffff0000  
alloc = 63   lookahead:     ffffff  
    fs blocks: ffffff0000  
scanning...  lookahead:     ffffff  
    fs blocks: ffffff0000  
scanning...  lookahead:     ffffff  
    fs blocks: ffffff0000  
scanning...  lookahead:     ffff0000  
    fs blocks: ffff0000  
alloc = 112   lookahead:         ffff8000  
    fs blocks: ffffff8000
```

While this lookahead approach still has an asymptotic runtime of $O(n^2)$ to scan all of storage, the lookahead reduces the practical runtime to a reasonable amount. Bit-vectors are surprisingly compact, given only 16 bytes, the lookahead could track 128 blocks. For a 4Mbyte flash chip with 4Kbyte blocks, the littlefs would only need 8 passes to scan the entire storage.

The real benefit of this approach is just how much it simplified the design of the littlefs. Deallocating blocks is as simple as simply forgetting they exist, and there is absolutely no concern of bugs in the deallocation code causing difficult to detect memory leaks.

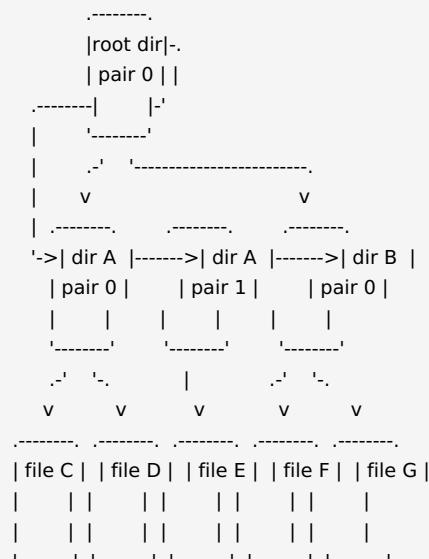
Directories

Now we just need directories to store our files. Since we already have metadata blocks that store information about files, lets just use these metadata blocks as the directories. Maybe turn the directories into linked lists of metadata blocks so it isn't limited by the number of files that fit in a single block. Add entries that represent other nested directories. Drop "." and ".." entries, cause who needs them. Dust off our hands and we now have a directory tree.



Unfortunately it turns out it's not that simple. See, iterating over a directory tree isn't actually all that easy, especially when you're trying to fit in a bounded amount of RAM, which rules out any recursive solution. And since our block allocator involves iterating over the entire filesystem tree, possibly multiple times in a single allocation, iteration needs to be efficient.

So, as a solution, the littlefs adopted a sort of threaded tree. Each directory not only contains pointers to all of its children, but also a pointer to the next directory. These pointers create a linked-list that is threaded through all of the directories in the filesystem. Since we only use this linked list to check for existence, the order doesn't actually matter. As an added plus, we can repurpose the pointer for the individual directory linked-lists and avoid using any additional space.



This threaded tree approach does come with a few tradeoffs. Now, anytime we want to manipulate the directory tree, we find ourselves having to update two pointers instead of one. For anyone familiar with creating atomic data structures this should set off a whole bunch of red flags.

But unlike the data structure guys, we can update a whole block atomically! So as long as we're really careful (and cheat a little bit), we can still manipulate the directory tree in a way that is resilient to power loss.

Consider how we might add a new directory. Since both pointers that reference it can come from the same directory, we only need a single atomic update to finagle the directory into the filesystem:

```
-----  
|root dir|.  
| pair 0 | |  
--|     |-'  
| '-----'  
|   |  
|   v  
| .-----.  
'->| dir A |  
| pair 0 |  
|   |  
'-----'  
  
| create the new directory block  
v
```

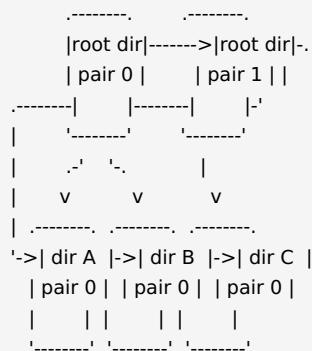
```
-----  
|root dir|.  
| pair 0 | |  
--|     |-'  
| '-----'  
|   |  
|   v  
| .-----.  
-----. '->| dir A |  
| dir B |--->| pair 0 |  
| pair 0 |   |   |  
|   |   '-----'  
'-----'
```

```
| update root to point to directory B  
v
```

```
-----  
|root dir|.  
| pair 0 | |  
-----|     |-'  
|   '-----'  
|   .-'   '-.  
|   v     v  
| .-----. .-----.  
'->| dir B |->| dir A |  
| pair 0 | | pair 0 |  
|   | |   |  
'-----' '-----'
```

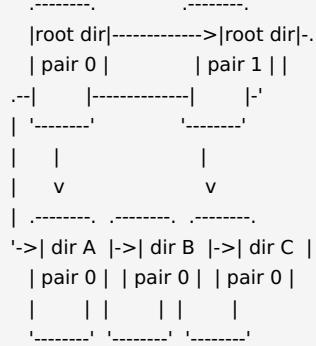
Note that even though directory B was added after directory A, we insert directory B before directory A in the linked-list because it is convenient.

Now how about removal:



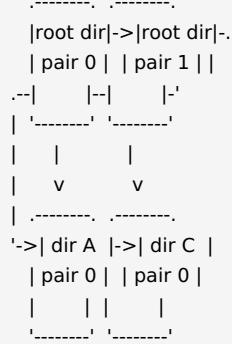
| update root to no longer contain directory B

v



| remove directory B from the linked-list

v



Wait, wait, wait, that's not atomic at all! If power is lost after removing directory B from the root, directory B is still in the linked-list. We've just created a memory leak!

And to be honest, I don't have a clever solution for this case. As a side-effect of using multiple pointers in the threaded tree, the littlefs can end up with orphan blocks that have no parents and should have been removed.

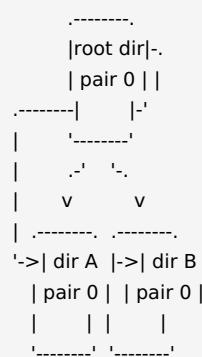
To keep these orphan blocks from becoming a problem, the littlefs has a deorphan step that simply iterates through every directory in the linked-list and checks it against every directory entry in the filesystem to see if it has a parent. The deorphan step occurs on the first block allocation after boot, so orphans should never cause the littlefs to run out of storage prematurely. Note that the deorphan step never needs to run in a read-only filesystem.

The move problem

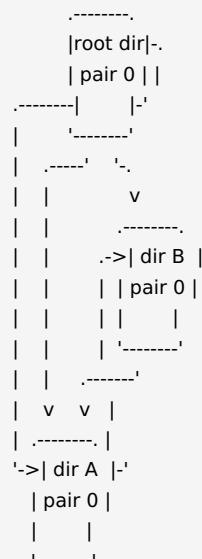
Now we have a real problem. How do we move things between directories while

remaining power resilient? Even looking at the problem from a high level, it seems impossible. We can update directory blocks atomically, but atomically updating two independent directory blocks is not an atomic operation.

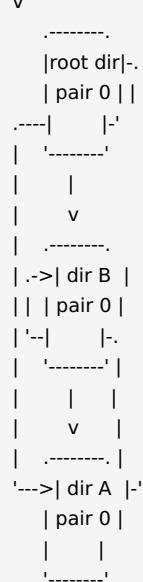
Here's the steps the filesystem may go through to move a directory:



| update directory B to point to directory A
v



| update root to no longer contain directory A
v



We can leave any orphans up to the deorphan step to collect, but that doesn't help the case where dir A has both dir B and the root dir as parents if we lose power inconveniently.

Initially, you might think this is fine. Dir A *might* end up with two parents,

but the filesystem will still work as intended. But then this raises the question of what do we do when the dir A wears out? For other directory blocks we can update the parent pointer, but for a dir with two parents we would need work out how to update both parents. And the check for multiple parents would need to be carried out for every directory, even if the directory has never been moved.

It also presents a bad user-experience, since the condition of ending up with two parents is rare, it's unlikely user-level code will be prepared. Just think about how a user would recover from a multi-parented directory. They can't just remove one directory, since remove would report the directory as "not empty".

Other atomic filesystems simple COW the entire directory tree. But this introduces a significant bit of complexity, which leads to code size, along with a surprisingly expensive runtime cost during what most users assume is a single pointer update.

Another option is to update the directory block we're moving from to point to the destination with a sort of predicate that we have moved if the destination exists. Unfortunately, the omnipresent concern of wear could cause any of these directory entries to change blocks, and changing the entry size before a move introduces complications if it spills out of the current directory block.

So how do we go about moving a directory atomically?

We rely on the improbableness of power loss.

Power loss during a move is certainly possible, but it's actually relatively rare. Unless a device is writing to a filesystem constantly, it's unlikely that a power loss will occur during filesystem activity. We still need to handle the condition, but runtime during a power loss takes a back seat to the runtime during normal operations.

So what littlefs does is inelegantly simple. When littlefs moves a file, it marks the file as "moving". This is stored as a single bit in the directory entry and doesn't take up much space. Then littlefs moves the directory, finishing with the complete remove of the "moving" directory entry.

```
-----.
|root dir|-
| pair 0 ||
-----|   |'
|       |
|       |
|       v   v
|       .-----.
'->| dir A |->| dir B |
| pair 0 | | pair 0 |
|       | |       |
'-----' '-----'

| update root directory to mark directory A as moving
v
```

```
-----.
|root dir |-
| pair 0 ||
-----| moving A!|-
|       |
|       |
|       v   v
|       .-----.
```

```
'->| dir A |->| dir B |
 | pair 0 | | pair 0 |
 |       | |       |
 '-----'
```

| update directory B to point to directory A

v

```
-----.
|root dir |-
| pair 0 |||
-----| moving A!|-'
| '-----'
| | .-.
| |   v
| |   -----
| |   .->| dir B | |
| |   | | pair 0 |
| |   | |   |
| |   | |   |
| |   | |   |
| |   | |   |
| |   | |   |
| |   v   v |
| |   -----|
'->| dir A |-'
| pair 0 |
|   |
'-----'
```

| update root to no longer contain directory A

v

```
-----.
|root dir|-
| pair 0 ||
-----|   |'-'
| '-----'
|   |
|   v
|   -----
| .->| dir B |
| | | pair 0 |
| '-|   |-
| '-----|'
|   |
|   v
|   -----
'--->| dir A |-'
| pair 0 |
|   |
'-----'
```

Now, if we run into a directory entry that has been marked as "moved", one of two things is possible. Either the directory entry exists elsewhere in the filesystem, or it doesn't. This is a O(n) operation, but only occurs in the unlikely case we lost power during a move.

And we can easily fix the "moved" directory entry. Since we're already scanning the filesystem during the deorphan step, we can also check for moved entries. If we find one, we either remove the "moved" marking or remove the whole entry if it exists elsewhere in the filesystem.

Wear awareness

So now that we have all of the pieces of a filesystem, we can look at a more subtle attribute of embedded storage: The wear down of flash blocks.

The first concern for the littlefs, is that perfectly valid blocks can suddenly

become unusable. As a nice side-effect of using a COW data-structure for files, we can simply move on to a different block when a file write fails. All modifications to files are performed in copies, so we will only replace the old file when we are sure none of the new file has errors. Directories, on the other hand, need a different strategy.

The solution to directory corruption in the littlefs relies on the redundant nature of the metadata pairs. If an error is detected during a write to one of the metadata pairs, we seek out a new block to take its place. Once we find a block without errors, we iterate through the directory tree, updating any references to the corrupted metadata pair to point to the new metadata block. Just like when we remove directories, we can lose power during this operation and end up with a desynchronized metadata pair in our filesystem. And just like when we remove directories, we leave the possibility of a desynchronized metadata pair up to the deorphan step to clean up.

Here's what encountering a directory error may look like with all of the directories and directory pointers fully expanded:

```
root dir
block 1  block 2
.-----.
| rev: 1 | rev: 0 |--.
|       |       |-'|
.----|       |       |'|
|---|       |       |'|
||  '-----'
||  |||||'-----.
||  |||||'-----.   |
||  ||||'-----.   |   |
||  ||'-----.   |   |   |
||  |'-----.   |   |   |
||  v       v       v       v       v       v
|| dir A     dir B     dir C
|| block 3  block 4  block 5  block 6  block 7  block 8
|| .-----. .-----. .-----.
|->| rev: 1 | rev: 0 |->| rev: 1 | rev: 0 |->| rev: 1 | rev: 0 |
'-->|       |       |->|       |       |->|       |       |
|       |       |       |       |       |       |
|       |       |       |       |       |       |
|       |       |       |       |       |       |
-----.
```

```
| update directory B
v
```

```
root dir
block 1  block 2
.-----.
| rev: 1 | rev: 0 |--.
|       |       |-'|
.----|       |       |'|
|---|       |       |'|
||  '-----'
||  |||||'-----.
||  |||||'-----.   |
||  ||||'-----.   |   |
||  ||'-----.   |   |   |
||  |'-----.   |   |   |
||  v       v       v       v       v       v
|| dir A     dir B     dir C
|| block 3  block 4  block 5  block 6  block 7  block 8
|| .-----. .-----. .-----.
|->| rev: 1 | rev: 0 |->| rev: 1 | rev: 2 |->| rev: 1 | rev: 0 |
'-->|       |       |->|       | corrupt!->|       |       |
|       |       |       | corrupt!|       |       |
|       |       |       | corrupt!|       |       |
```

| oh no! corruption detected
v allocate a replacement block

root dir

block 1 block 2

```

-----.
| rev: 1 | rev: 0 |--.
|   |   | .|
-----|   |   | |
|.---|   |   | |
||   |-----|
||   ||||'-----.
||   ||||'-----.
||   ||||'-----.
||   ||'-----.
||   |-----.
||   v   v   v   v   v   v
|| dir A     dir B     dir C
|| block 3   block 4   block 5   block 6   block 7   block 8
|| -----, -----, -----, -----
!->| rev: 1 | rev: 0 |->| rev: 1 | rev: 2 |--->| rev: 1 | rev: 0
'-->|   |   | ->|   | corrupt!|->|   |   |
|   |   |   |   | corrupt!| .->|   |   |
|   |   |   |   | corrupt!|   |   |
-----'-----'-----'-----'
                                block 9 |
-----.
| rev: 2 |-'|
|   |   |
|   |   |
|   |   |
-----'
```

| update root directory to contain block 9

V

```

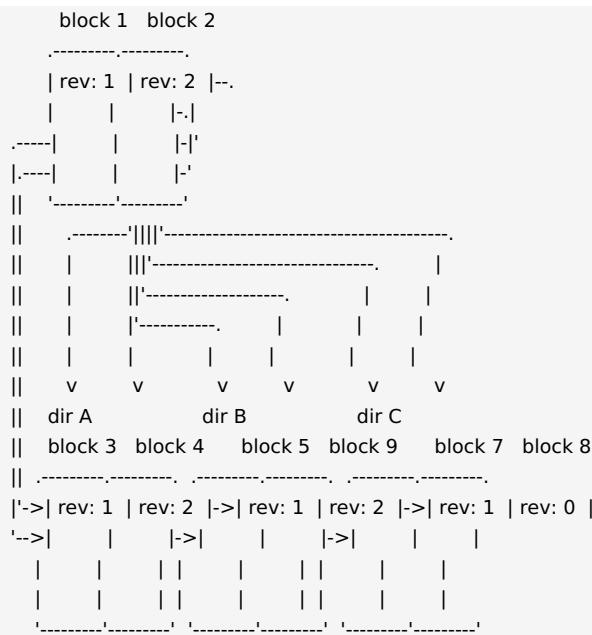
root dir
block 1  block 2
-----.
| rev: 1 | rev: 2 |--.
|     |     |.-|
|     |     |-'|
|     |     |-'|
||  -----'|
||  .----'|'||'-----.
||  |  ||'-----.
||  |  ||'-----.
||  |  |'-----.
||  |  |   |   |   |
||  v  v   v   v   v   v
|| dir A      dir B      dir C
|| block 3    block 4    block 5    block 9    block 7    block 8
|| -----.  .-----.  .-----.  .-----.
|->| rev: 1 | rev: 0 |-->| rev: 1 | -| rev: 2 |--->| rev: 1 | rev: 0
'-->|     |     |.-|     |   |   |.->|     |     |
|     |     |   |     |   |   |   |     |     |
|     |     |   |     |   |   |   |     |     |
|-----'|   '|   '|   '|   '|   '|-----'
|           block 6 |
|           .-----|
|----->| rev: 2 |-
| corrupt!|
| corrupt!|
| corrupt!|

```

I remove corrupted block from linked-list

16

root dir



Also one question I've been getting is, what about the root directory? It can't move so wouldn't the filesystem die as soon as the root blocks develop errors? And you would be correct. So instead of storing the root in the first few blocks of the storage, the root is actually pointed to by the superblock. The superblock contains a few bits of static data, but outside of when the filesystem is formatted, it is only updated when the root develops errors and needs to be moved.

Wear leveling

The second concern for the littlefs is that blocks in the filesystem may wear unevenly. In this situation, a filesystem may meet an early demise where there are no more non-corrupted blocks that aren't in use. It's common to have files that were written once and left unmodified, wasting the potential erase cycles of the blocks it sits on.

Wear leveling is a term that describes distributing block writes evenly to avoid the early termination of a flash part. There are typically two levels of wear leveling:

1. Dynamic wear leveling - Wear is distributed evenly across all **dynamic** blocks. Usually this is accomplished by simply choosing the unused block with the lowest amount of wear. Note this does not solve the problem of static data.
2. Static wear leveling - Wear is distributed evenly across all **dynamic** and **static** blocks. Unmodified blocks may be evicted for new block writes. This does handle the problem of static data but may lead to wear amplification.

In littlefs's case, it's possible to use the revision count on metadata pairs to approximate the wear of a metadata block. And combined with the COW nature of files, littlefs could provide your usual implementation of dynamic wear leveling.

However, the littlefs does not. This is for a few reasons. Most notably, even if the littlefs did implement dynamic wear leveling, this would still not handle the case of write-once files, and near the end of the lifetime of a flash device, you would likely end up with uneven wear on the blocks anyways.

As a flash device reaches the end of its life, the metadata blocks will naturally be the first to go since they are updated most often. In this situation, the littlefs is designed to simply move on to another set of metadata blocks. This travelling means that at the end of a flash device's life, the filesystem will have worn the device down nearly as evenly as the usual dynamic wear leveling could. More aggressive wear leveling would come with a code-size cost for marginal benefit.

One important takeaway to note, if your storage stack uses highly sensitive storage such as NAND flash, static wear leveling is the only valid solution. In most cases you are going to be better off using a full [flash translation layer \(FTL\)](#).

NAND flash already has many limitations that make it poorly suited for an embedded system: low erase cycles, very large blocks, errors that can develop even during reads, errors that can develop during writes of neighboring blocks. Managing sensitive storage such as NAND flash is out of scope for the littlefs. The littlefs does have some properties that may be beneficial on top of a FTL, such as limiting the number of writes where possible, but if you have the storage requirements that necessitate the need of NAND flash, you should have the RAM to match and just use an FTL or flash filesystem.

Summary

So, to summarize:

1. The littlefs is composed of directory blocks
2. Each directory is a linked-list of metadata pairs
3. These metadata pairs can be updated atomically by alternating which metadata block is active
4. Directory blocks contain either references to other directories or files
5. Files are represented by copy-on-write CTZ skip-lists which support O(1) append and O(n log n) reading
6. Blocks are allocated by scanning the filesystem for used blocks in a fixed-size lookahead region that is stored in a bit-vector
7. To facilitate scanning the filesystem, all directories are part of a linked-list that is threaded through the entire filesystem
8. If a block develops an error, the littlefs allocates a new block, and moves the data and references of the old block to the new.
9. Any case where an atomic operation is not possible, mistakes are resolved by a deorphan step that occurs on the first allocation after boot

That's the little filesystem. Thanks for reading!

Copyright (c) 2017, Arm Limited. All rights reserved.

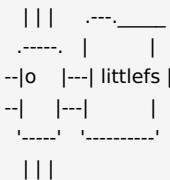
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of ARM nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The little filesystem

A little fail-safe filesystem designed for embedded systems.



Bounded RAM/ROM - The littlefs is designed to work with a limited amount of memory. Recursion is avoided and dynamic memory is limited to configurable buffers that can be provided statically.

Power-loss resilient - The littlefs is designed for systems that may have random power failures. The littlefs has strong copy-on-write guarantees and storage on disk is always kept in a valid state.

Wear leveling - Since the most common form of embedded storage is erodible flash memories, littlefs provides a form of dynamic wear leveling for systems that can not fit a full flash translation layer.

Example

Here's a simple example that updates a file named `boot_count` every time main runs. The program can be interrupted at any time without losing track of how many times it has been booted and without corrupting the filesystem:

```

#include "lfs1.h"

// variables used by the filesystem
lfs1_t lfs1;
lfs1_file_t file;

// configuration of the filesystem is provided by this struct
const struct lfs1_config cfg = {
    // block device operations
    .read   = user_provided_block_device_read,
    .prog   = user_provided_block_device_prog,
    .erase   = user_provided_block_device_erase,
    .sync   = user_provided_block_device_sync,

    // block device configuration
    .read_size = 16,
    .prog_size = 16,
    .block_size = 4096,
    .block_count = 128,
    .lookahead = 128,
};

// entry point
int main(void) {
    // mount the filesystem
    int err = lfs1_mount(&lfs1, &cfg);

    // reformat if we can't mount the filesystem
    // this should only happen on the first boot
    if (err) {
        lfs1_format(&lfs1, &cfg);
        lfs1_mount(&lfs1, &cfg);
    }

    // read current count
    uint32_t boot_count = 0;
    lfs1_file_open(&lfs1, &file, "boot_count", LFS1_O_RDWR | LFS1_O_CREAT);
    lfs1_file_read(&lfs1, &file, &boot_count, sizeof(boot_count));

    // update boot count
    boot_count += 1;
    lfs1_file_rewind(&lfs1, &file);
    lfs1_file_write(&lfs1, &file, &boot_count, sizeof(boot_count));

    // remember the storage is not updated until the file is closed successfully
    lfs1_file_close(&lfs1, &file);

    // release any resources we were using
    lfs1_unmount(&lfs1);

    // print the boot count
    printf("boot_count: %d\n", boot_count);
}

```

Usage

Detailed documentation (or at least as much detail as is currently available) can be found in the comments in [lfs1.h](#).

As you may have noticed, littlefs takes in a configuration structure that defines how the filesystem operates. The configuration struct provides the filesystem with the block device operations and dimensions, tweakable parameters that tradeoff memory usage for performance, and optional static buffers if the user wants to avoid dynamic memory.

The state of the littlefs is stored in the `lfs1_t` type which is left up

to the user to allocate, allowing multiple filesystems to be in use simultaneously. With the `lfs1_t` and configuration struct, a user can format a block device or mount the filesystem.

Once mounted, the littlefs provides a full set of POSIX-like file and directory functions, with the deviation that the allocation of filesystem structures must be provided by the user.

All POSIX operations, such as remove and rename, are atomic, even in event of power-loss. Additionally, no file updates are actually committed to the filesystem until sync or close is called on the file.

Other notes

All littlefs calls have the potential to return a negative error code. The errors can be either one of those found in the `enum lfs1_error` in `lfs1.h`, or an error returned by the user's block device operations.

In the configuration struct, the `prog` and `erase` function provided by the user may return a `LFS1_ERR_CORRUPT` error if the implementation already can detect corrupt blocks. However, the wear leveling does not depend on the return code of these functions, instead all data is read back and checked for integrity.

If your storage caches writes, make sure that the provided `sync` function flushes all the data to memory and ensures that the next read fetches the data from memory, otherwise data integrity can not be guaranteed. If the `write` function does not perform caching, and therefore each `read` or `write` call hits the memory, the `sync` function can simply return 0.

Reference material

[DESIGN.md](#) - [DESIGN.md](#) contains a fully detailed dive into how littlefs actually works. I would encourage you to read it since the solutions and tradeoffs at work here are quite interesting.

[SPEC.md](#) - [SPEC.md](#) contains the on-disk specification of littlefs with all the nitty-gritty details. Can be useful for developing tooling.

Testing

The littlefs comes with a test suite designed to run on a PC using the [emulated block device](#) found in the emubd directory.

The tests assume a Linux environment and can be started with make:

```
make test
```

License

The littlefs is provided under the [BSD-3-Clause](#) license. See [LICENSE.md](#) for more information. Contributions to this project are accepted under the same license.

Individual files contain the following tag instead of the full license text.

This enables machine processing of license information based on the SPDX License Identifiers that are here available: <http://spdx.org/licenses/>

Related projects

Mbed OS -

The easiest way to get started with littlefs is to jump into [Mbed](#), which already has block device drivers for most forms of embedded storage. The littlefs is available in Mbed OS as the [LittleFileSystem](#) class.

[littlefs-fuse](#) - A FUSE

wrapper for littlefs. The project allows you to mount littlefs directly on a Linux machine. Can be useful for debugging littlefs if you have an SD card handy.

[littlefs-js](#) - A javascript wrapper for littlefs. I'm not sure why you would want this, but it is handy for demos. You can see it in action [here](#).

[mk-lfs](#) -

A command line tool built by the [Lua RTOS](#) guys for making littlefs images from a host PC. Supports Windows, Mac OS, and Linux.

[SPIFFS](#) - Another excellent embedded filesystem for NOR flash. As a more traditional logging filesystem with full static wear-leveling, SPIFFS will likely outperform littlefs on small memories such as the internal flash on microcontrollers.

[Dhara](#) - An interesting NAND flash translation layer designed for small MCUs. It offers static wear-leveling and power-resilience with only a fixed $O(|address|)$ pointer structure stored on each block and in RAM.

The little filesystem technical specification

This is the technical specification of the little filesystem. This document covers the technical details of how the littlefs is stored on disk for introspection and tooling development. This document assumes you are familiar with the design of the littlefs, for more info on how littlefs works check out [DESIGN.md](#).

```
  |||  .--._____
  .----. |   |
--|o  |---| littlefs |
--|  |---|   |
'----' '-----'
  |||
```

Some important details

- The littlefs is a block-based filesystem. This is, the disk is divided into an array of evenly sized blocks that are used as the logical unit of storage

in littlefs. Block pointers are stored in 32 bits.

- There is no explicit free-list stored on disk, the littlefs only knows what is in use in the filesystem.
- The littlefs uses the value of 0xffffffff to represent a null block-pointer.
- All values in littlefs are stored in little-endian byte order.

Directories / Metadata pairs

Metadata pairs form the backbone of the littlefs and provide a system for atomic updates. Even the superblock is stored in a metadata pair.

As their name suggests, a metadata pair is stored in two blocks, with one block acting as a redundant backup in case the other is corrupted. These two blocks could be anywhere in the disk and may not be next to each other, so any pointers to directory pairs need to be stored as two block pointers.

Here's the layout of metadata blocks on disk:

offset	size	description
0x00	32 bits	revision count
0x04	32 bits	dir size
0x08	64 bits	tail pointer
0x10	size-16 bytes	dir entries
0x00+s	32 bits	CRC

Revision count - Incremented every update, only the uncorrupted metadata-block with the most recent revision count contains the valid metadata. Comparison between revision counts must use sequence comparison since the revision counts may overflow.

Dir size - Size in bytes of the contents in the current metadata block, including the metadata-pair metadata. Additionally, the highest bit of the dir size may be set to indicate that the directory's contents continue on the next metadata-pair pointed to by the tail pointer.

Tail pointer - Pointer to the next metadata-pair in the filesystem.

A null pair-pointer (0xffffffff, 0xffffffff) indicates the end of the list. If the highest bit in the dir size is set, this points to the next metadata-pair in the current directory, otherwise it points to an arbitrary metadata-pair. Starting with the superblock, the tail-pointers form a linked-list containing all metadata-pairs in the filesystem.

CRC - 32 bit CRC used to detect corruption from power-lost, from block end-of-life, or just from noise on the storage bus. The CRC is appended to the end of each metadata-block. The littlefs uses the standard CRC-32, which uses a polynomial of 0x04c11db7, initialized with 0xffffffff.

Here's an example of a simple directory stored on disk:

```
(32 bits) revision count = 10          (0x0000000a)
(32 bits) dir size      = 154 bytes, end of dir (0x0000009a)
(64 bits) tail pointer  = 37, 36       (0x00000025, 0x00000024)
(32 bits) CRC           = 0xc86e3106
```

```
00000000: 0a 00 00 00 9a 00 00 00 25 00 00 00 24 00 00 00 .....%....$...
00000010: 22 08 00 03 05 00 00 00 04 00 00 00 74 65 61 22 ".....tea"
00000020: 08 00 06 07 00 00 00 06 00 00 00 63 6f 66 66 65 .....coffe
00000030: 65 22 08 00 04 09 00 00 00 08 00 00 00 73 6f 64 e".....sod
00000040: 61 22 08 00 05 1d 00 00 00 1c 00 00 00 6d 69 6c a".....mil
00000050: 6b 31 22 08 00 05 1f 00 00 00 1e 00 00 00 6d 69 k1".....mi
00000060: 6c 6b 32 22 08 00 05 21 00 00 00 20 00 00 00 6d lk2"!...!....m
00000070: 69 6c 6b 33 22 08 00 05 23 00 00 00 22 00 00 00 ilk3"#!..."...
00000080: 6d 69 6c 6b 34 22 08 00 05 25 00 00 00 24 00 00 milk4"!%...$..
00000090: 00 6d 69 6c 6b 35 06 31 6e c8 .milk5.1n.
```

A note about the tail pointer linked-list: Normally, this linked-list is threaded through the entire filesystem. However, after power-loss this linked-list may become out of sync with the rest of the filesystem.

- The linked-list may contain a directory that has actually been removed
- The linked-list may contain a metadata pair that has not been updated after a block in the pair has gone bad.

The threaded linked-list must be checked for these errors before it can be used reliably. Fortunately, the threaded linked-list can simply be ignored if littlefs is mounted read-only.

Entries

Each metadata block contains a series of entries that follow a standard layout. An entry contains the type of the entry, along with a section for entry-specific data, attributes, and a name.

Here's the layout of entries on disk:

offset	size	description
0x0	8 bits	entry type
0x1	8 bits	entry length
0x2	8 bits	attribute length
0x3	8 bits	name length
0x4	entry length bytes	entry-specific data
0x4+e	attribute length bytes	system-specific attributes
0x4+e+a	name length bytes	entry name

Entry type - Type of the entry, currently this is limited to the following:

- 0x11 - file entry
- 0x22 - directory entry
- 0x2e - superblock entry

Additionally, the type is broken into two 4 bit nibbles, with the upper nibble specifying the type's data structure used when scanning the filesystem. The lower nibble clarifies the type further when multiple entries share the same data structure.

The highest bit is reserved for marking the entry as "moved". If an entry is marked as "moved", the entry may also exist somewhere else in the filesystem. If the entry exists elsewhere, this entry must be treated as though it does not exist.

Entry length - Length in bytes of the entry-specific data. This does not include the entry type size, attributes, or name. The full size in bytes of the entry is $4 + \text{entry length} + \text{attribute length} + \text{name length}$.

Attribute length - Length of system-specific attributes in bytes. Since attributes are system specific, there is not much guarantee on the values in this section, and systems are expected to work even when it is empty. See the [attributes](#) section for more details.

Name length - Length of the entry name. Entry names are stored as UTF8, although most systems will probably only support ASCII. Entry names can not contain '/' and can not be '.' or '..' as these are a part of the syntax of filesystem paths.

Here's an example of a simple entry stored on disk:

```
(8 bits) entry type      = file    (0x11)
(8 bits) entry length    = 8 bytes (0x08)
(8 bits) attribute length = 0 bytes (0x00)
(8 bits) name length     = 12 bytes (0x0c)
(8 bytes) entry data      = 05 00 00 00 20 00 00 00
(12 bytes) entry name     = smallavocado

00000000: 11 08 00 0c 05 00 00 00 20 00 00 00 73 6d 61 6c ..... .smal
00000010: 6c 61 76 61 63 61 64 6f           lavacado
```

Superblock

The superblock is the anchor for the littlefs. The superblock is stored as a metadata pair containing a single superblock entry. It is through the superblock that littlefs can access the rest of the filesystem.

The superblock can always be found in blocks 0 and 1, however fetching the superblock requires knowing the block size. The block size can be guessed by searching the beginning of disk for the string "littlefs", although currently the filesystems relies on the user providing the correct block size.

The superblock is the most valuable block in the filesystem. It is updated very rarely, only during format or when the root directory must be moved. It is encouraged to always write out both superblock pairs even though it is not required.

Here's the layout of the superblock entry:

offset	size	description
0x00	8 bits	entry type (0x2e for superblock entry)
0x01	8 bits	entry length (20 bytes)
0x02	8 bits	attribute length
0x03	8 bits	name length (8 bytes)
0x04	64 bits	root directory
0x0c	32 bits	block size

offset	size	block size
0x10	32 bits	block count
0x14	32 bits	version
0x18	attribute length bytes	system-specific attributes
0x18+a	8 bytes	magic string ("littlefs")

Root directory - Pointer to the root directory's metadata pair.

Block size - Size of the logical block size used by the filesystem.

Block count - Number of blocks in the filesystem.

Version - The littlefs version encoded as a 32 bit value. The upper 16 bits encodes the major version, which is incremented when a breaking-change is introduced in the filesystem specification. The lower 16 bits encodes the minor version, which is incremented when a backwards-compatible change is introduced. Non-standard Attribute changes do not change the version. This specification describes version 1.1 (0x00010001), which is the first version of littlefs.

Magic string - The magic string "littlefs" takes the place of an entry name.

Here's an example of a complete superblock:

```
(32 bits) revision count = 3          (0x00000003)
(32 bits) dir size     = 52 bytes, end of dir (0x00000034)
(64 bits) tail pointer = 3, 2         (0x00000003, 0x00000002)
(8 bits) entry type   = superblock   (0x2e)
(8 bits) entry length  = 20 bytes    (0x14)
(8 bits) attribute length = 0 bytes  (0x00)
(8 bits) name length   = 8 bytes    (0x08)
(64 bits) root directory = 3, 2       (0x00000003, 0x00000002)
(32 bits) block size    = 512 bytes   (0x00000200)
(32 bits) block count   = 1024 blocks (0x00000400)
(32 bits) version      = 1.1        (0x00010001)
(8 bytes) magic string  = littlefs
(32 bits) CRC          = 0xc50b74fa

00000000: 03 00 00 00 34 00 00 00 03 00 00 00 02 00 00 00 ....4.....
00000010: 2e 14 00 08 03 00 00 00 02 00 00 00 00 02 00 00 .....
00000020: 00 04 00 00 01 00 01 00 6c 69 74 74 6c 65 66 73 .....littlefs
00000030: fa 74 0b c5               .t..
```

Directory entries

Directories are stored in entries with a pointer to the first metadata pair in the directory. Keep in mind that a directory may be composed of multiple metadata pairs connected by the tail pointer when the highest bit in the dir size is set.

Here's the layout of a directory entry:

offset	size	description
0x0	8 bits	entry type (0x22 for directory entries)
0x1	8 bits	entry length (8 bytes)
0x2	8 bits	attribute length

0x3	8 bits	name length
0x4	64 bits	directory pointer
0xc	attribute length bytes	system-specific attributes
0xc+a	name length bytes	directory name

Directory pointer - Pointer to the first metadata pair in the directory.

Here's an example of a directory entry:

```
(8 bits) entry type      = directory (0x22)
(8 bits) entry length    = 8 bytes  (0x08)
(8 bits) attribute length = 0 bytes (0x00)
(8 bits) name length     = 3 bytes (0x03)
(64 bits) directory pointer = 5, 4   (0x00000005, 0x00000004)
(3 bytes) name           = tea

00000000: 22 08 00 03 05 00 00 00 04 00 00 00 74 65 61  ".....tea
```

File entries

Files are stored in entries with a pointer to the head of the file and the size of the file. This is enough information to determine the state of the CTZ skip-list that is being referenced.

How files are actually stored on disk is a bit complicated. The full explanation of CTZ skip-lists can be found in [DESIGN.md](#).

A terribly quick summary: For every nth block where n is divisible by 2^x , the block contains a pointer to block $n-2^x$. These pointers are stored in increasing order of x in each block of the file preceding the data in the block.

The maximum number of pointers in a block is bounded by the maximum file size divided by the block size. With 32 bits for file size, this results in a minimum block size of 104 bytes.

Here's the layout of a file entry:

offset	size	description
0x0	8 bits	entry type (0x11 for file entries)
0x1	8 bits	entry length (8 bytes)
0x2	8 bits	attribute length
0x3	8 bits	name length
0x4	32 bits	file head
0x8	32 bits	file size
0xc	attribute length bytes	system-specific attributes
0xc+a	name length bytes	directory name

File head - Pointer to the block that is the head of the file's CTZ skip-list.

File size - Size of file in bytes.

Here's an example of a file entry:

```
(8 bits) entry type      = file    (0x11)
(8 bits) entry length    = 8 bytes (0x08)
(8 bits) attribute length = 0 bytes (0x00)
(8 bits) name length     = 12 bytes (0x03)
(32 bits) file head      = 543    (0x00000021f)
(32 bits) file size       = 256 KB (0x00040000)
(12 bytes) name          = largeavacado

00000000: 11 08 00 0c 1f 02 00 00 00 00 04 00 6c 61 72 67 .....larg
00000010: 65 61 76 61 63 61 64 6f           eavacado
```

Entry attributes

Each dir entry can have up to 256 bytes of system-specific attributes. Since these attributes are system-specific, they may not be portable between different systems. For this reason, all attributes must be optional. A minimal littlefs driver must be able to get away with supporting no attributes at all.

For some level of portability, littlefs has a simple scheme for attributes. Each attribute is prefixes with an 8-bit type that indicates what the attribute is. The length of attributes may also be determined from this type. Attributes in an entry should be sorted based on portability, since attribute parsing will end when it hits the first attribute it does not understand.

Each system should choose a 4-bit value to prefix all attribute types with to avoid conflicts with other systems. Additionally, littlefs drivers that support attributes should provide a "ignore attributes" flag to users in case attribute conflicts do occur.

Attribute types prefixes with 0x0 and 0xf are currently reserved for future standard attributes. Standard attributes will be added to this document in that case.

Here's an example of non-standard time attribute:

```
(8 bits) attribute type = time    (0xc1)
(72 bits) time in seconds = 1506286115 (0x0059c81a23)

00000000: c1 23 1a c8 59 00           .#..Y.
```

Here's an example of non-standard permissions attribute:

```
(8 bits) attribute type = permissions (0xc2)
(16 bits) permission bits = rw-rw-r-- (0x01b4)

00000000: c2 b4 01           ...
```

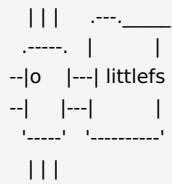
Here's what a dir entry may look like with these attributes:

```
(8 bits) entry type      = file      (0x11)
(8 bits) entry length    = 8 bytes    (0x08)
(8 bits) attribute length = 9 bytes    (0x09)
(8 bits) name length     = 12 bytes   (0x0c)
(8 bytes) entry data     = 05 00 00 00 20 00 00 00
(8 bits) attribute type  = time      (0xc1)
(72 bits) time in seconds = 1506286115 (0x0059c81a23)
(8 bits) attribute type  = permissions (0xc2)
(16 bits) permission bits = rw-rw-r-- (0x01b4)
(12 bytes) entry name    = smallavacado
```

```
00000000: 11 08 09 0c 05 00 00 00 20 00 00 00 c1 23 1a c8 ..... ....#..
00000010: 59 00 c2 b4 01 73 6d 61 6c 61 76 61 63 61 64 Y....smallavacad
00000020: 6f          o
```

The design of littlefs

A little fail-safe filesystem designed for microcontrollers.



littlefs was originally built as an experiment to learn about filesystem design in the context of microcontrollers. The question was: How would you build a filesystem that is resilient to power-loss and flash wear without using unbounded memory?

This document covers the high-level design of littlefs, how it is different than other filesystems, and the design decisions that got us here. For the low-level details covering every bit on disk, check out [SPEC.md](#).

The problem

The embedded systems littlefs targets are usually 32-bit microcontrollers with around 32 KiB of RAM and 512 KiB of ROM. These are often paired with SPI NOR flash chips with about 4 MiB of flash storage. These devices are too small for Linux and most existing filesystems, requiring code written specifically with size in mind.

Flash itself is an interesting piece of technology with its own quirks and nuance. Unlike other forms of storage, writing to flash requires two operations: erasing and programming. Programming (setting bits to 0) is relatively cheap and can be very granular. Erasing however (setting bits to 1), requires an expensive and destructive operation which gives flash its name. [Wikipedia](#) has more information on how exactly flash works.

To make the situation more annoying, it's very common for these embedded systems to lose power at any time. Usually, microcontroller code is simple and reactive, with no concept of a shutdown routine. This presents a big challenge for persistent storage, where an unlucky power loss can corrupt the storage and leave a device unrecoverable.

This leaves us with three major requirements for an embedded filesystem.

1. **Power-loss resilience** - On these systems, power can be lost at any time.

If a power loss corrupts any persistent data structures, this can cause the device to become unrecoverable. An embedded filesystem must be designed to recover from a power loss during any write operation.

2. **Wear leveling** - Writing to flash is destructive. If a filesystem repeatedly writes to the same block, eventually that block will wear out. Filesystems that don't take wear into account can easily burn through blocks used to store frequently updated metadata and cause a device's early death.

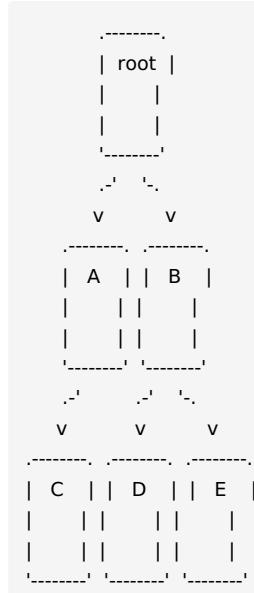
3. **Bounded RAM/ROM** - If the above requirements weren't enough, these systems also have very limited amounts of memory. This prevents many existing filesystem designs, which can lean on relatively large amounts of RAM to temporarily store filesystem metadata.

For ROM, this means we need to keep our design simple and reuse code paths were possible. For RAM we have a stronger requirement, all RAM usage is bounded. This means RAM usage does not grow as the filesystem changes in size or number of files. This creates a unique challenge as even presumably simple operations, such as traversing the filesystem, become surprisingly difficult.

Existing designs?

So, what's already out there? There are, of course, many different filesystems, however they often share and borrow feature from each other. If we look at power-loss resilience and wear leveling, we can narrow these down to a handful of designs.

1. First we have the non-resilient, block based filesystems, such as [FAT](#) and [ext2](#). These are the earliest filesystem designs and often the most simple. Here storage is divided into blocks, with each file being stored in a collection of blocks. Without modifications, these filesystems are not power-loss resilient, so updating a file is as simple as rewriting the blocks in place.

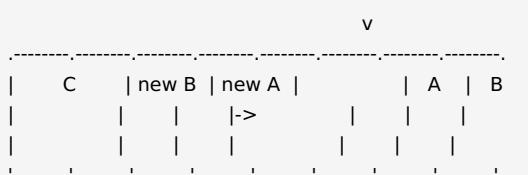


Because of their simplicity, these filesystems are usually both the fastest and smallest. However the lack of power resilience is not great, and the binding relationship of storage location and data removes the filesystem's ability to manage wear.

2. In a completely different direction, we have logging filesystems, such as [JFFS](#), [YAFFS](#), and [SPIFFS](#), storage location is not bound to a piece of data, instead the entire storage is used for a circular log which is

appended with every change made to the filesystem. Writing appends new changes, while reading requires traversing the log to reconstruct a file.

Some logging filesystems cache files to avoid the read cost, but this comes at a tradeoff of RAM.



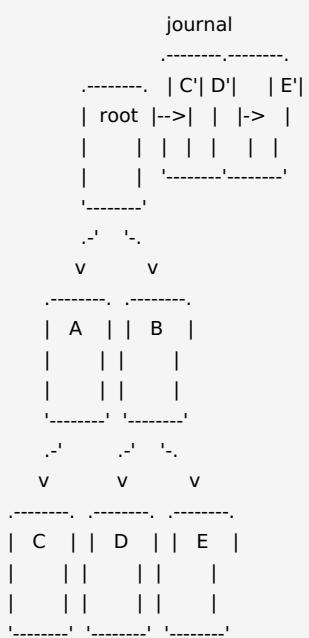
Logging filesystems are beautifully elegant. With a checksum, we can easily detect power-loss and fall back to the previous state by ignoring failed appends. And if that wasn't good enough, their cyclic nature means that logging filesystems distribute wear across storage perfectly.

The main downside is performance. If we look at garbage collection, the process of cleaning up outdated data from the end of the log, I've yet to see a pure logging filesystem that does not have one of these two costs:

1. $O(n^2)$ runtime
2. $O(n)$ RAM

SPIFFS is a very interesting case here, as it uses the fact that repeated programs to NOR flash is both atomic and masking. This is a very neat solution, however it limits the type of storage you can support.

3. Perhaps the most common type of filesystem, a journaling filesystem is the offspring that happens when you mate a block based filesystem with a logging filesystem. [ext4](#) and [NTFS](#) are good examples. Here, we take a normal block based filesystem and add a bounded log where we note every change before it occurs.

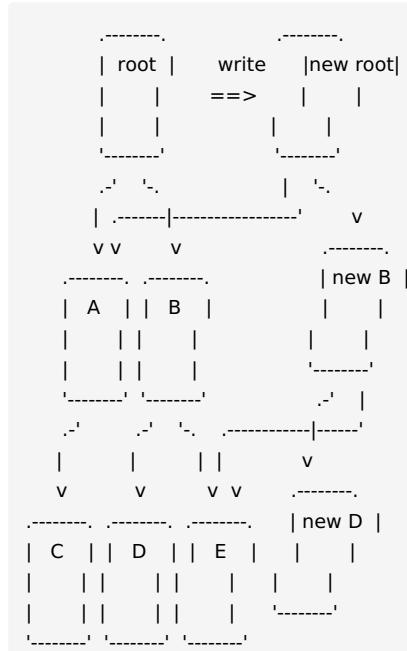


This sort of filesystem takes the best from both worlds. Performance can be as fast as a block based filesystem (though updating the journal does have a small cost), and atomic updates to the journal allow the filesystem to recover in the event of a power loss.

Unfortunately, journaling filesystems have a couple of problems. They are fairly complex, since there are effectively two filesystems running in parallel, which comes with a code size cost. They also offer no protection against wear because of the strong relationship between storage location and data.

4. Last but not least we have copy-on-write (COW) filesystems, such as

btrfs and **ZFS**. These are very similar to other block based filesystems, but instead of updating block in place, all updates are performed by creating a copy with the changes and replacing any references to the old block with our new block. This recursively pushes all of our problems upwards until we reach the root of our filesystem, which is often stored in a very small log.



COW filesystems are interesting. They offer very similar performance to block based filesystems while managing to pull off atomic updates without storing data changes directly in a log. They even disassociate the storage location of data, which creates an opportunity for wear leveling.

Well, almost. The unbounded upwards movement of updates causes some problems. Because updates to a COW filesystem don't stop until they've reached the root, an update can cascade into a larger set of writes than would be needed for the original data. On top of this, the upward motion focuses these writes into the block, which can wear out much earlier than the rest of the filesystem.

littlefs

So what does littlefs do?

If we look at existing filesystems, there are two interesting design patterns that stand out, but each have their own set of problems. Logging, which provides independent atomicity, has poor runtime performance. And COW data structures, which perform well, push the atomicity problem upwards.

Can we work around these limitations?

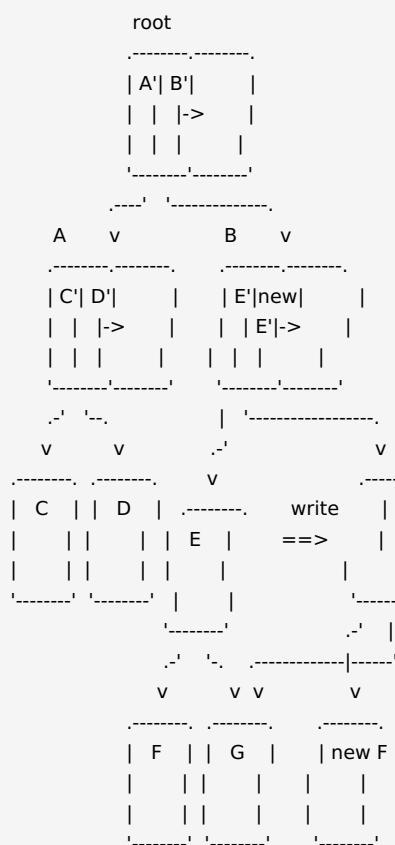
Consider logging. It has either a $O(n^2)$ runtime or $O(n)$ RAM cost. We can't avoid these costs, *but* if we put an upper bound on the size we can at least prevent the theoretical cost from becoming problem. This relies on the super secret computer science hack where you can pretend any algorithmic complexity is $O(1)$ by bounding the input.

In the case of COW data structures, we can try twisting the definition a bit. Let's say that our COW structure doesn't copy after a single write, but instead copies after n writes. This doesn't change most COW properties (assuming you can write atomically!), but what it does do is prevent the upward motion of wear. This sort of copy-on-bounded-writes (COBW) still focuses wear, but at

each level we divide the propagation of wear by n . With a sufficiently large n ($>$ branching factor) wear propagation is no longer a problem.

See where this is going? Separate, logging and COW are imperfect solutions and have weaknesses that limit their usefulness. But if we merge the two they can mutually solve each other's limitations.

This is the idea behind littlefs. At the sub-block level, littlefs is built out of small, two block logs that provide atomic updates to metadata anywhere on the filesystem. At the super-block level, littlefs is a CObW tree of blocks that can be evicted on demand.



There are still some minor issues. Small logs can be expensive in terms of storage, in the worst case a small log costs 4x the size of the original data. CObW structures require an efficient block allocator since allocation occurs every n writes. And there is still the challenge of keeping the RAM usage constant.

Metadata pairs

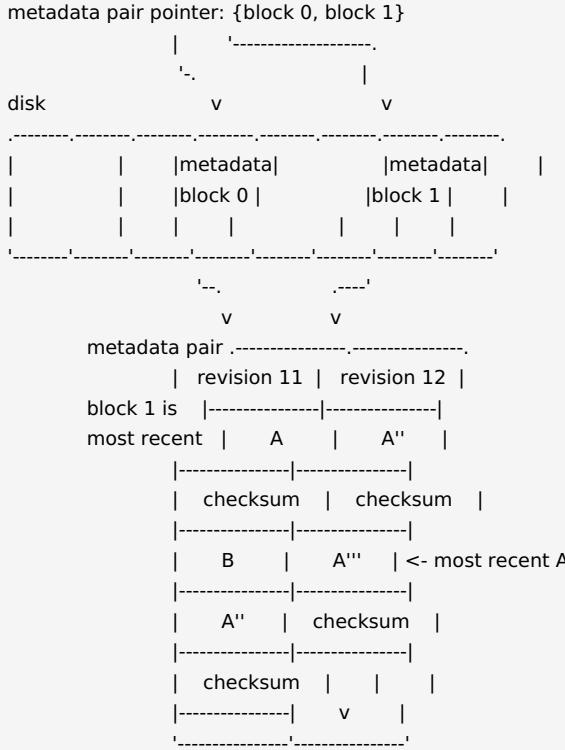
Metadata pairs are the backbone of littlefs. These are small, two block logs that allow atomic updates anywhere in the filesystem.

Why two blocks? Well, logs work by appending entries to a circular buffer stored on disk. But remember that flash has limited write granularity. We can incrementally program new data onto erased blocks, but we need to erase a full block at a time. This means that in order for our circular buffer to work, we need more than one block.

We could make our logs larger than two blocks, but the next challenge is how do we store references to these logs? Because the blocks themselves are erased during writes, using a data structure to track these blocks is complicated. The simple solution here is to store a two block addresses for every metadata

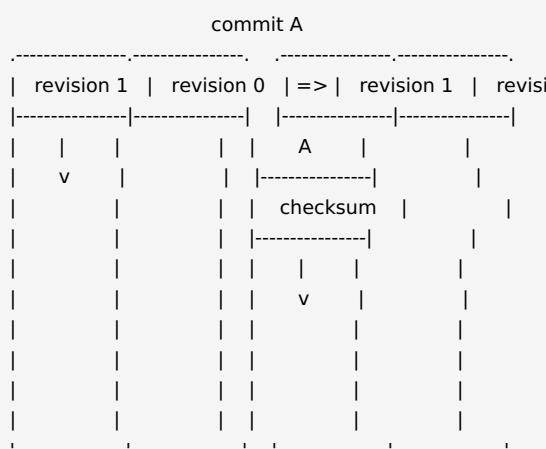
pair. This has the added advantage that we can change out blocks in the metadata pair independently, and we don't reduce our block granularity for other operations.

In order to determine which metadata block is the most recent, we store a revision count that we compare using [sequence arithmetic](#) (very handy for avoiding problems with integer overflow). Conveniently, this revision count also gives us a rough idea of how many erases have occurred on the block.



So how do we atomically update our metadata pairs? Atomicity (a type of power-loss resilience) requires two parts: redundancy and error detection. Error detection can be provided with a checksum, and in littlefs's case we use a 32-bit **CRC**. Maintaining redundancy, on the other hand, requires multiple stages.

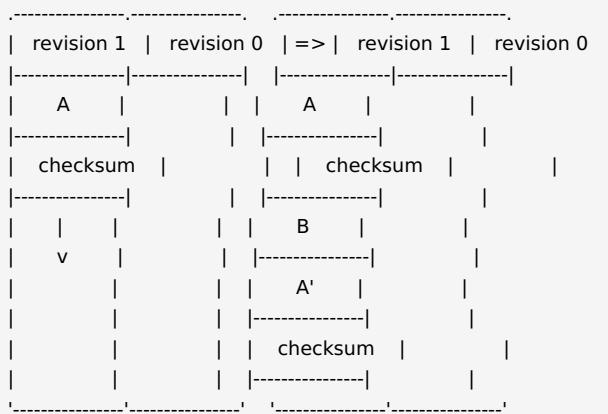
1. If our block is not full and the program size is small enough to let us append more entries, we can simply append the entries to the log. Because we don't overwrite the original entries (remember rewriting flash requires an erase), we still have the original entries if we lose power during the append.



Note that littlefs doesn't maintain a checksum for each entry. Many logging

filesystems do this, but it limits what you can update in a single atomic operation. What we can do instead is group multiple entries into a commit that shares a single checksum. This lets us update multiple unrelated pieces of metadata as long as they reside on the same metadata pair.

commit B and A'

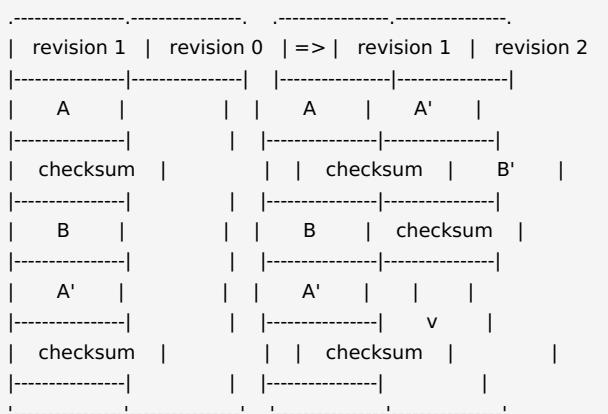


2. If our block *is* full of entries, we need to somehow remove outdated entries to make space for new ones. This process is called garbage collection, but because littlefs has multiple garbage collectors, we also call this specific case compaction.

Compared to other filesystems, littlefs's garbage collector is relatively simple. We want to avoid RAM consumption, so we use a sort of brute force solution where for each entry we check to see if a newer entry has been written. If the entry is the most recent we append it to our new block. This is where having two blocks becomes important, if we lose power we still have everything in our original block.

During this compaction step we also erase the metadata block and increment the revision count. Because we can commit multiple entries at once, we can write all of these changes to the second block without worrying about power loss. It's only when the commit's checksum is written that the compacted entries and revision count become committed and readable.

commit B', need to compact



3. If our block is full of entries *and* we can't find any garbage, then what?

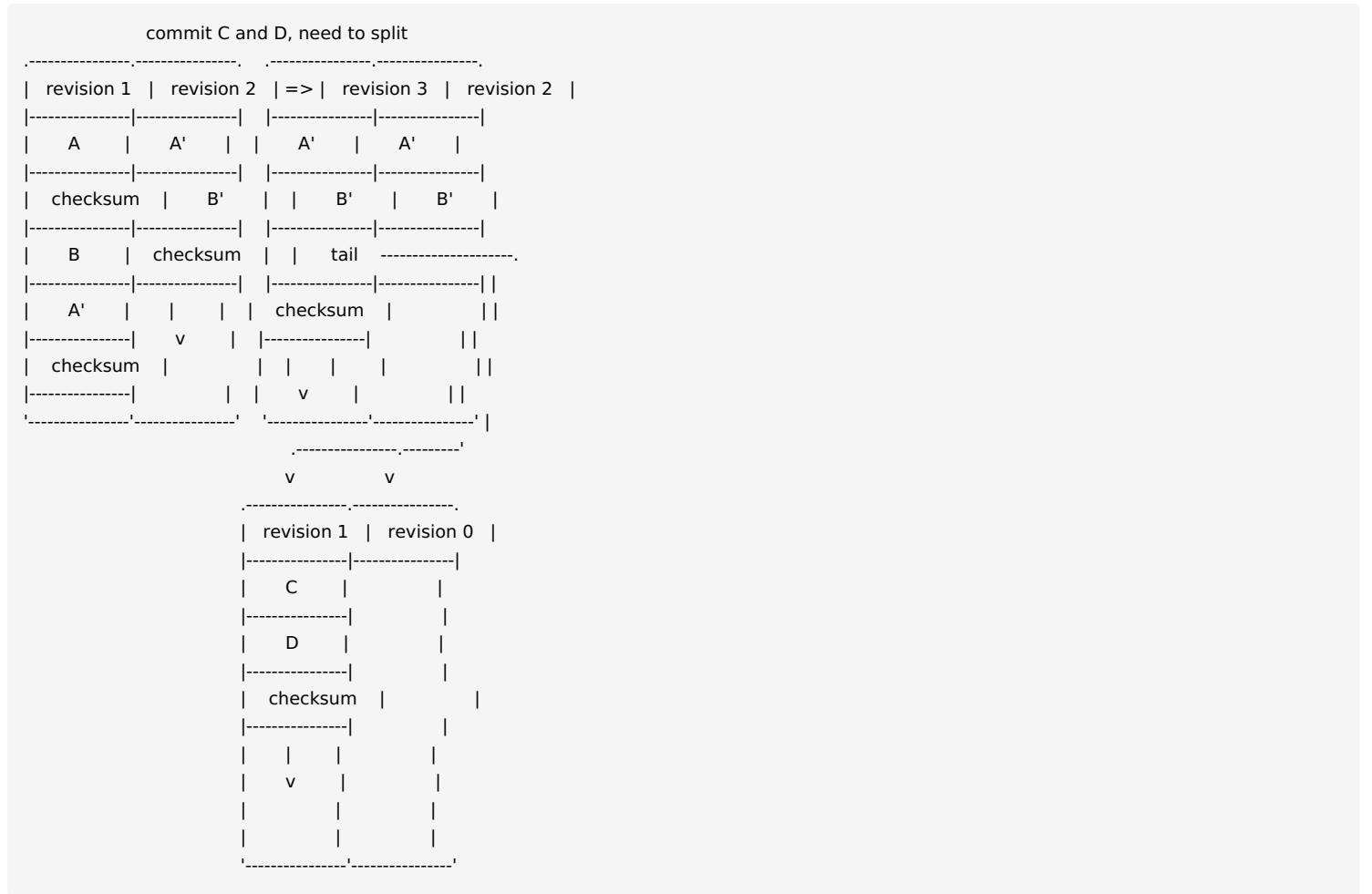
At this point, most logging filesystems would return an error indicating no more space is available, but because we have small logs, overflowing a log isn't really an error condition.

Instead, we split our original metadata pair into two metadata pairs, each containing half of the entries, connected by a tail pointer. Instead of increasing the size of the log and dealing with the scalability issues associated with larger logs, we form a linked list of small bounded logs.

This is a tradeoff as this approach does use more storage space, but at the

benefit of improved scalability.

Despite writing to two metadata pairs, we can still maintain power resilience during this split step by first preparing the new metadata pair, and then inserting the tail pointer during the commit to the original metadata pair.



There is another complexity that crops up when dealing with small logs. The amortized runtime cost of garbage collection is not only dependent on its one time cost ($O(n^2)$ for littlefs), but also depends on how often garbage collection occurs.

Consider two extremes:

1. Log is empty, garbage collection occurs once every n updates
2. Log is full, garbage collection occurs **every** update

Clearly we need to be more aggressive than waiting for our metadata pair to be full. As the metadata pair approaches fullness the frequency of compactions grows very rapidly.

Looking at the problem generically, consider a log with bytes for each entry, dynamic entries (entries that are outdated during garbage collection), and static entries (entries that need to be copied during garbage collection). If we look at the amortized runtime complexity of updating this log we get this formula:

If we let

be the ratio of static space to the size of our log in bytes, we

find an alternative representation of the number of static and dynamic entries:

Substituting these in for

and

gives us a nice formula for the cost of

updating an entry given how full the log is:

Assuming 100 byte entries in a 4 KiB log, we can graph this using the entry size to find a multiplicative cost:

So at 50% usage, we're seeing an average of 2x cost per update, and at 75% usage, we're already at an average of 4x cost per update.

To avoid this exponential growth, instead of waiting for our metadata pair to be full, we split the metadata pair once we exceed 50% capacity. We do this lazily, waiting until we need to compact before checking if we fit in our 50% limit. This limits the overhead of garbage collection to 2x the runtime cost, giving us an amortized runtime complexity of $O(1)$.

If we look at metadata pairs and linked-lists of metadata pairs at a high level, they have fairly nice runtime costs. Assuming n metadata pairs, each containing m metadata entries, the *lookup* cost for a specific entry has a worst case runtime complexity of $O(nm)$. For *updating* a specific entry, the worst case complexity is $O(nm^2)$, with an amortized complexity of only $O(nm)$.

However, splitting at 50% capacity does mean that in the best case our metadata pairs will only be 1/2 full. If we include the overhead of the second block in our metadata pair, each metadata entry has an effective storage cost of 4x the original size. I imagine users would not be happy if they found that they can only use a quarter of their original storage. Metadata pairs provide a mechanism for performing atomic updates, but we need a separate mechanism for storing the bulk of our data.

CTZ skip-lists

Metadata pairs provide efficient atomic updates but unfortunately have a large storage cost. But we can work around this storage cost by only using the metadata pairs to store references to more dense, copy-on-write (COW) data structures.

[Copy-on-write data structures](#), also called purely functional data structures, are a category of data structures where the underlying elements are immutable. Making changes to the data requires creating new elements containing a copy of the updated data and replacing any references with references to the new elements. Generally, the performance of a COW data structure depends on how many old elements can be reused after replacing parts of the data.

littlefs has several requirements of its COW structures. They need to be efficient to read and write, but most frustrating, they need to be traversable with a constant amount of RAM. Notably this rules out

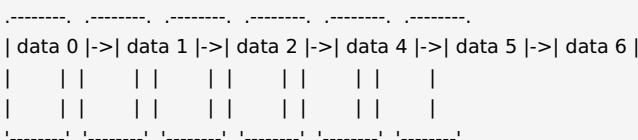
[B-trees](#), which can not be traversed with constant RAM, and

[B+-trees](#), which are not possible to update with COW

operations.

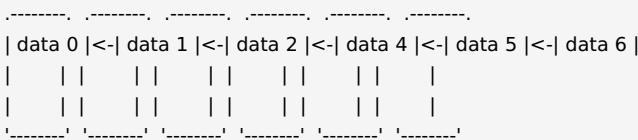
So, what can we do? First let's consider storing files in a simple COW linked-list. Appending a block, which is the basis for writing files, means we have to update the last block to point to our new block. This requires a COW operation, which means we need to update the second-to-last block, and then the third-to-last, and so on until we've copied out the entire file.

A linked-list



To avoid a full copy during appends, we can store the data backwards. Appending blocks just requires adding the new block and no other blocks need to be updated. If we update a block in the middle, we still need to copy the following blocks, but can reuse any blocks before it. Since most file writes are linear, this design gambles that appends are the most common type of data update.

A backwards linked-list



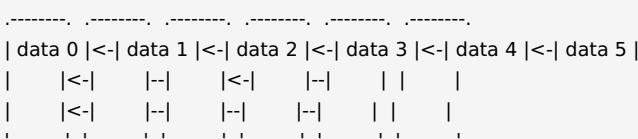
However, a backwards linked-list does have a rather glaring problem. Iterating over a file *in order* has a runtime cost of $O(n^2)$. A quadratic runtime just to read a file! That's awful.

Fortunately we can do better. Instead of a singly linked list, littlefs uses a multilayered linked-list often called a [skip-list](#). However, unlike the most common type of skip-list, littlefs's skip-lists are strictly deterministic built around some interesting properties of the count-trailing-zeros (CTZ) instruction.

The rules CTZ skip-lists follow are that for every n th block where n is divisible by 2^x , that block contains a pointer to block $n-2^x$. This means that each block contains anywhere from 1 to $\log_2 n$ pointers that skip to different preceding elements of the skip-list.

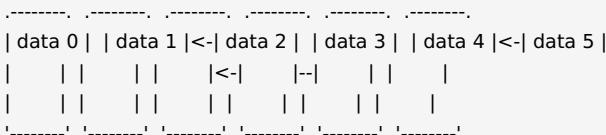
The name comes from heavy use of the [CTZ instruction](#), which lets us calculate the power-of-two factors efficiently. For a give block n , that block contains $\text{ctz}(n)+1$ pointers.

A backwards CTZ skip-list

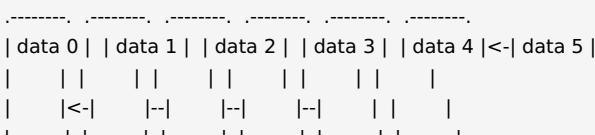


The additional pointers let us navigate the data-structure on disk much more efficiently than in a singly linked list.

Consider a path from data block 5 to data block 1. You can see how data block 3 was completely skipped:



The path to data block 0 is even faster, requiring only two jumps:



We can find the runtime complexity by looking at the path to any block from the block containing the most pointers. Every step along the path divides the search space for the block in half, giving us a runtime of $O(\log n)$.

To get to the block with the most pointers, we can perform the same steps backwards, which puts the runtime at $O(2 \log n) = O(\log n)$. An interesting note is that this optimal path occurs naturally if we greedily choose the pointer that covers the most distance without passing our target.

So now we have a [COW](#) data structure that is cheap to append with a runtime of $O(1)$, and can be read with a worst case runtime of $O(n \log n)$. Given that this runtime is also divided by the amount of data we can store in a block, this cost is fairly reasonable.

This is a new data structure, so we still have several questions. What is the storage overhead? Can the number of pointers exceed the size of a block? How do we store a CTZ skip-list in our metadata pairs?

To find the storage overhead, we can look at the data structure as multiple linked-lists. Each linked-list skips twice as many blocks as the previous, or from another perspective, each linked-list uses half as much storage as the previous. As we approach infinity, the storage overhead forms a geometric series. Solving this tells us that on average our storage overhead is only 2 pointers per block.

Because our file size is limited the word width we use to store sizes, we can also solve for the maximum number of pointers we would ever need to store in a block. If we set the overhead of pointers equal to the block size, we get the following equation. Note that both a smaller block size () and larger word width () result in more storage overhead.

Solving the equation for gives us the minimum block size for some common word widths:

1. 32-bit CTZ skip-list => minimum block size of 104 bytes
2. 64-bit CTZ skip-list => minimum block size of 448 bytes

littlefs uses a 32-bit word width, so our blocks can only overflow with pointers if they are smaller than 104 bytes. This is an easy requirement, as in practice, most block sizes start at 512 bytes. As long as our block size is larger than 104 bytes, we can avoid the extra logic needed to handle pointer overflow.

This last question is how do we store CTZ skip-lists? We need a pointer to the head block, the size of the skip-list, the index of the head block, and our offset in the head block. But it's worth noting that each size maps to a unique index + offset pair. So in theory we can store only a single pointer and size.

However, calculating the index + offset pair from the size is a bit complicated. We can start with a summation that loops through all of the blocks up until our given size. Let
be the block size in bytes,
be the
word width in bits,
be the index of the block in the skip-list, and
be the file size in bytes:

This works quite well, but requires $O(n)$ to compute, which brings the full runtime of reading a file up to $O(n^2 \log n)$. Fortunately, that summation doesn't need to touch the disk, so the practical impact is minimal.

However, despite the integration of a bitwise operation, we can actually reduce this equation to a $O(1)$ form. While browsing the amazing resource that is the [On-Line Encyclopedia of Integer Sequences \(OEIS\)](#), I managed to find [A001511](#), which matches the iteration of the CTZ instruction, and [A005187](#), which matches its partial summation. Much to my surprise, these both result from simple equations, leading us to a rather unintuitive property that ties together two seemingly unrelated bitwise instructions:

where:

1. `ctz()`
) = the number of trailing bits that are 0 in
2. `popcount()`
) = the number of bits that are 1 in

Initial tests of this surprising property seem to hold. As approaches infinity, we end up with an average overhead of 2 pointers, which matches our assumption from earlier. During iteration, the popcount function seems to handle deviations from this average. Of course, just to make sure I wrote a quick script that verified this property for all 32-bit integers.

Now we can substitute into our original equation to find a more efficient equation for file size:

Unfortunately, the popcount function is non-injective, so we can't solve this equation for our index. But what we can do is solve for an index that

is greater than

with error bounded by the range of the `popcount` function.

We can repeatedly substitute

into the original equation until the error

is smaller than our integer resolution. As it turns out, we only need to perform this substitution once, which gives us this formula for our index:

Now that we have our index

, we can just plug it back into the above

equation to find the offset. We run into a bit of a problem with integer overflow, but we can avoid this by rearranging the equation a bit:

Our solution requires quite a bit of math, but computers are very good at math.

Now we can find both our block index and offset from a size in $O(1)$, letting

us store CTZ skip-lists with only a pointer and size.

CTZ skip-lists give us a COW data structure that is easily traversable in $O(n)$, can be appended in $O(1)$, and can be read in $O(n \log n)$. All of these operations work in a bounded amount of RAM and require only two words of storage overhead per block. In combination with metadata pairs, CTZ skip-lists provide power resilience and compact storage of data.

```

-----.
.|metadata|
||   |
||   |
|'-----'
'---|---'
      v
-----.
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 | | |
|   |<-|   |-|   ||   |
|   ||   ||   ||   ||   |
'-----' '-----' '-----'

write data to disk, create copies
=>

-----.
.|metadata|
||   |
||   |
|'-----'
'---|---'
      v
-----.
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 | | |
|   |<-|   |-|   ||   |
|   ||   ||   ||   ||   |
'-----' '-----' '-----'

^ ^   ^
||   | ----- .----- .
||   '---| new  |<-| new  |<-| new  |<-| new  |
| '-----| data 2 |<-| data 3 |--| data 4 | | data 5 |
'-----|   |-|   |-|   ||   |
      '-----' '-----'

commit to metadata pair
=>

-----.
.|new   |
|||metadata|
||   |
|'-----'
'---|---'
      |
-----.
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 | | |
|   |<-|   |-|   ||   |
|   ||   ||   ||   ||   |
'-----' '-----' '-----'

^ ^   ^           v
||   | ----- .----- .
||   '---| new  |<-| new  |<-| new  |<-| new  |
| '-----| data 2 |<-| data 3 |--| data 4 | | data 5 |
'-----|   |-|   |-|   ||   |
      '-----' '-----'

```

The block allocator

So we now have the framework for an atomic, wear leveling filesystem. Small two block metadata pairs provide atomic updates, while CTZ skip-lists provide compact storage of data in COW blocks.

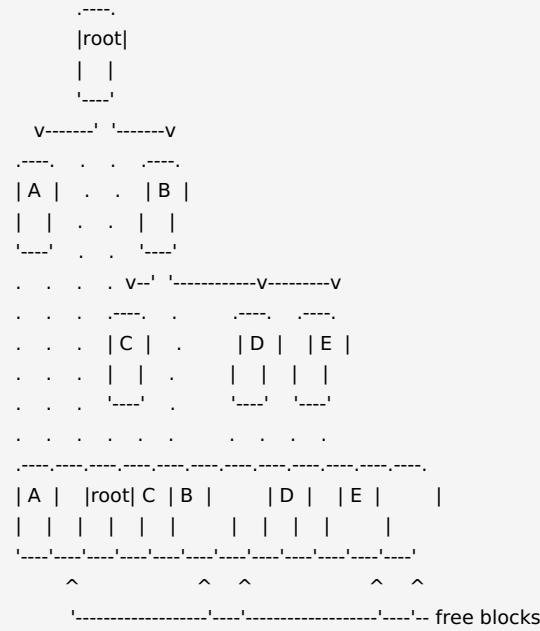
But now we need to look at the [elephant](#) in the room. Where do all these blocks come from?

Deciding which block to use next is the responsibility of the block allocator. In filesystem design, block allocation is often a second-class citizen, but in a COW filesystem its role becomes much more important as it is needed for

nearly every write to the filesystem.

Normally, block allocation involves some sort of free list or bitmap stored on the filesystem that is updated with free blocks. However, with power resilience, keeping these structures consistent becomes difficult. It doesn't help that any mistake in updating these structures can result in lost blocks that are impossible to recover.

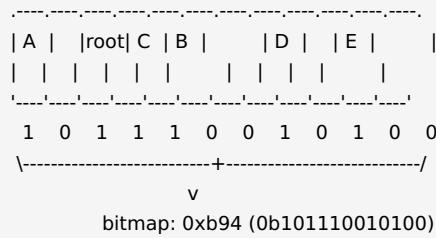
littlefs takes a cautious approach. Instead of trusting a free list on disk, littlefs relies on the fact that the filesystem on disk is a mirror image of the free blocks on the disk. The block allocator operates much like a garbage collector in a scripting language, scanning for unused blocks on demand.



While this approach may sound complicated, the decision to not maintain a free list greatly simplifies the overall design of littlefs. Unlike programming languages, there are only a handful of data structures we need to traverse. And block deallocation, which occurs nearly as often as block allocation, is simply a noop. This "drop it on the floor" strategy greatly reduces the complexity of managing on disk data structures, especially when handling high-risk error conditions.

Our block allocator needs to find free blocks efficiently. You could traverse through every block on storage and check each one against our filesystem tree; however, the runtime would be abhorrent. We need to somehow collect multiple blocks per traversal.

Looking at existing designs, some larger filesystems that use a similar "drop it on the floor" strategy store a bitmap of the entire storage in **RAM**. This works well because bitmaps are surprisingly compact. We can't use the same strategy here, as it violates our constant RAM requirement, but we may be able to modify the idea into a workable solution.



The block allocator in littlefs is a compromise between a disk-sized bitmap and a brute force traversal. Instead of a bitmap the size of storage, we keep track of a small, fixed-size bitmap called the lookahead buffer. During block allocation, we take blocks from the lookahead buffer. If the lookahead buffer is empty, we scan the filesystem for more free blocks, populating our lookahead buffer. In each scan we use an increasing offset, circling the storage as blocks are allocated.

Here's what it might look like to allocate 4 blocks on a decently busy filesystem with a 32 bit lookahead and a total of 128 blocks (512 KiB of storage if blocks are 4 KiB):

```

boot...      lookahead:
           fs blocks: fffff9fffffffffffffeffffffffffff0000
scanning...   lookahead: fffff9ff
           fs blocks: fffff9fffffffffffffeffffffffffff0000
alloc = 21    lookahead: fffffdff
           fs blocks: fffffdfffffffffffffeffffffffffff0000
alloc = 22    lookahead: ffffffff
           fs blocks: ffffffffffffeffffffffffff0000
scanning...   lookahead:     ffffffe
           fs blocks: ffffffffffffeffffffffffff0000
alloc = 63    lookahead:     fffffff
           fs blocks: ffffffffffffeffffffffffff0000
scanning...   lookahead:     ffffffff
           fs blocks: ffffffffffffeffffffffffff0000
scanning...   lookahead:     ffffffrf
           fs blocks: ffffffffffffeffffffffffff0000
scanning...   lookahead:             ffff0000
           fs blocks: ffffffffffffeffffffffffff0000
alloc = 112   lookahead:             ffff8000
           fs blocks: ffffffffffffeffffffffffff8000

```

This lookahead approach has a runtime complexity of $O(n^2)$ to completely scan storage; however, bitmaps are surprisingly compact, and in practice only one or two passes are usually needed to find free blocks. Additionally, the performance of the allocator can be optimized by adjusting the block size or size of the lookahead buffer, trading either write granularity or RAM for allocator performance.

Wear leveling

The block allocator has a secondary role: wear leveling.

Wear leveling is the process of distributing wear across all blocks in the storage to prevent the filesystem from experiencing an early death due to wear on a single block in the storage.

littlefs has two methods of protecting against wear:

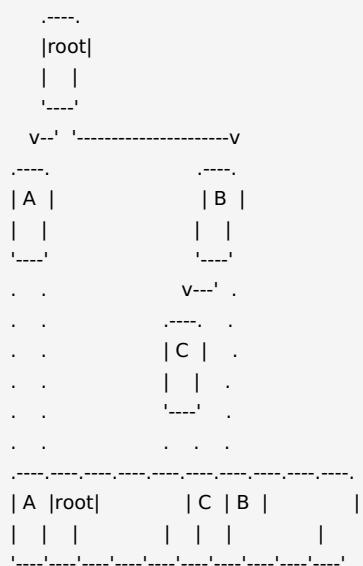
1. Detection and recovery from bad blocks
2. Evenly distributing wear across dynamic blocks

Recovery from bad blocks doesn't actually have anything to do with the block allocator itself. Instead, it relies on the ability of the filesystem to detect and evict bad blocks when they occur.

In littlefs, it is fairly straightforward to detect bad blocks at write time. All writes must be sourced by some form of data in RAM, so immediately after we write to a block, we can read the data back and verify that it was written correctly. If we find that the data on disk does not match the copy we have in RAM, a write error has occurred and we most likely have a bad block.

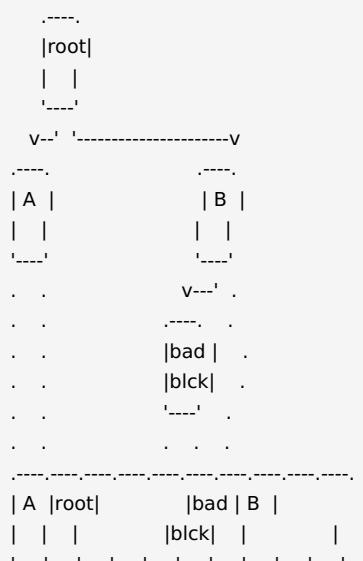
Once we detect a bad block, we need to recover from it. In the case of write errors, we have a copy of the corrupted data in RAM, so all we need to do is evict the bad block, allocate a new, hopefully good block, and repeat the write that previously failed.

The actual act of evicting the bad block and replacing it with a new block is left up to the filesystem's copy-on-bounded-writes (COBW) data structures. One property of COBW data structures is that any block can be replaced during a COW operation. The bounded-writes part is normally triggered by a counter, but nothing prevents us from triggering a COW operation as soon as we find a bad block.



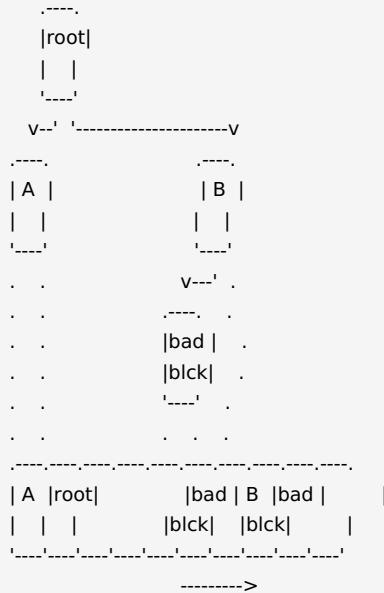
update C

=>



oh no! bad block! relocate C

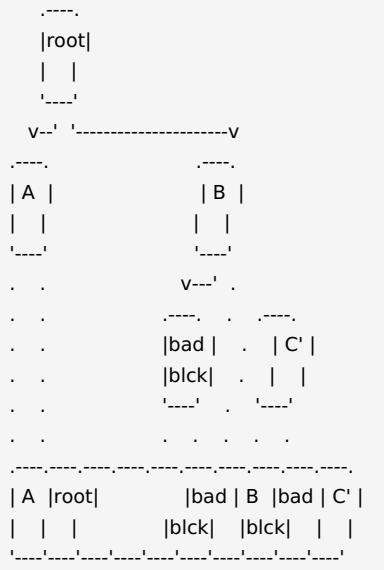
=>



----->

oh no! bad block! relocate C

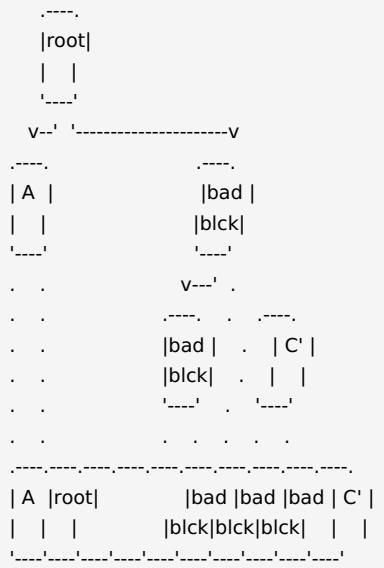
=>



----->

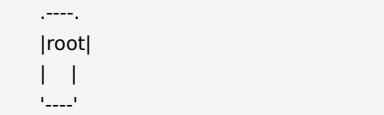
successfully relocated C, update B

=>



oh no! bad block! relocate B

=>



```

v--' -----v
----. . . .
| A |     |bad|     |bad|
|_ _|     |blk|     |blk|
----' . . .
. . . . . . .
. . . |bad| . . |C'|
. . . |blk| . . | |
. . . '---' . . .
. . . . . . .
-----.
| A |root|     |bad|bad|bad| C' |bad|
|_ _|     |blk|blk|blk|   |blk|
----->

```

oh no! bad block! relocate B

=>

```

----.
|root|
|_ _|
----'
v--' -----v
----. . . .
| A | | B' |     |bad|
|_ _| |_ |     |blk|
----' . . .
. . . | . . . .
. . . . . . .
. . . . |bad| . . |C'|
. . . . |blk| . . | |
. . . . '---' . . .
. . . . . . .
-----.
| A |root| B' |     |bad|bad|bad| C' |bad|
|_ _|     |blk|blk|blk|   |blk|
----->

```

successfully relocated B, update root

=>

```

----.
|root|
|_ _|
----'
v--' --v
----. . . .
| A | | B' |
|_ _| |_ |
----' . . .
. . . . . . .
. . . . | C' |
. . . . | |
. . . . '---' . . .
. . . . . . .
-----.
| A |root| B' |     |bad|bad|bad| C' |bad|
|_ _|     |blk|blk|blk|   |blk|
-----'
```

We may find that the new block is also bad, but hopefully after repeating this cycle we'll eventually find a new block where a write succeeds. If we don't, that means that all blocks in our storage are bad, and we've reached the end of our device's usable life. At this point, littlefs will return an "out of space" error. This is technically true, as there are no more good blocks, but as an added benefit it also matches the error condition expected by users of dynamically sized data.

Read errors, on the other hand, are quite a bit more complicated. We don't have a copy of the data lingering around in RAM, so we need a way to reconstruct the original data even after it has been corrupted. One such mechanism for this is [error-correction-codes \(ECC\)](#).

ECC is an extension to the idea of a checksum. Where a checksum such as CRC can detect that an error has occurred in the data, ECC can detect and actually correct some amount of errors. However, there is a limit to how many errors ECC can detect: the [Hamming bound](#). As the number of errors approaches the Hamming bound, we may still be able to detect errors, but can no longer fix the data. If we've reached this point the block is unrecoverable.

littlefs by itself does **not** provide ECC. The block nature and relatively large footprint of ECC does not work well with the dynamically sized data of filesystems, correcting errors without RAM is complicated, and ECC fits better with the geometry of block devices. In fact, several NOR flash chips have extra storage intended for ECC, and many NAND chips can even calculate ECC on the chip itself.

In littlefs, ECC is entirely optional. Read errors can instead be prevented proactively by wear leveling. But it's important to note that ECC can be used at the block device level to modestly extend the life of a device. littlefs respects any errors reported by the block device, allowing a block device to provide additional aggressive error detection.

To avoid read errors, we need to be proactive, as opposed to reactive as we were with write errors.

One way to do this is to detect when the number of errors in a block exceeds some threshold, but is still recoverable. With ECC we can do this at write time, and treat the error as a write error, evicting the block before fatal read errors have a chance to develop.

A different, more generic strategy, is to proactively distribute wear across all blocks in the storage, with the hope that no single block fails before the rest of storage is approaching the end of its usable life. This is called wear leveling.

Generally, wear leveling algorithms fall into one of two categories:

1. [Dynamic wear leveling](#), where we distribute wear over "dynamic" blocks. This can be accomplished by only considering unused blocks.
2. [Static wear leveling](#), where we distribute wear over both "dynamic" and "static" blocks. To make this work, we need to consider all blocks, including blocks that already contain data.

As a tradeoff for code size and complexity, littlefs (currently) only provides dynamic wear leveling. This is a best effort solution. Wear is not distributed perfectly, but it is distributed among the free blocks and greatly extends the life of a device.

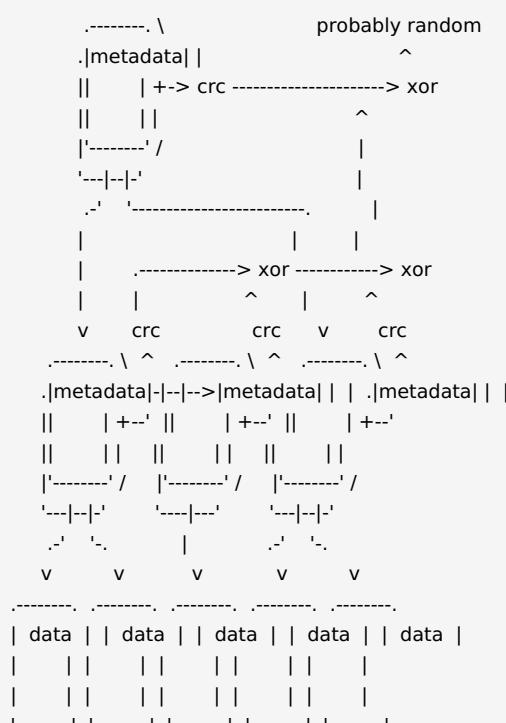
On top of this, littlefs uses a statistical wear leveling algorithm. What this

means is that we don't actively track wear, instead we rely on a uniform distribution of wear across storage to approximate a dynamic wear leveling algorithm. Despite the long name, this is actually a simplification of dynamic wear leveling.

The uniform distribution of wear is left up to the block allocator, which creates a uniform distribution in two parts. The easy part is when the device is powered, in which case we allocate the blocks linearly, circling the device. The harder part is what to do when the device loses power. We can't just restart the allocator at the beginning of storage, as this would bias the wear. Instead, we start the allocator as a random offset every time we mount the filesystem. As long as this random offset is uniform, the combined allocation pattern is also a uniform distribution.

Initially, this approach to wear leveling looks like it creates a difficult dependency on a power-independent random number generator, which must return different random numbers on each boot. However, the filesystem is in a relatively unique situation in that it is sitting on top of a large amount of entropy that persists across power loss.

We can actually use the data on disk to directly drive our random number generator. In practice, this is implemented by xor-ing the checksums of each metadata pair, which is already calculated to fetch and mount the filesystem.



Note that this random number generator is not perfect. It only returns unique random numbers when the filesystem is modified. This is exactly what we want for distributing wear in the allocator, but means this random number generator is not useful for general use.

Together, bad block detection and dynamic wear leveling provide a best effort solution for avoiding the early death of a filesystem due to wear. Importantly, littlefs's wear leveling algorithm provides a key feature: You can increase the life of a device simply by increasing the size of storage. And if more aggressive wear leveling is desired, you can always combine littlefs with a

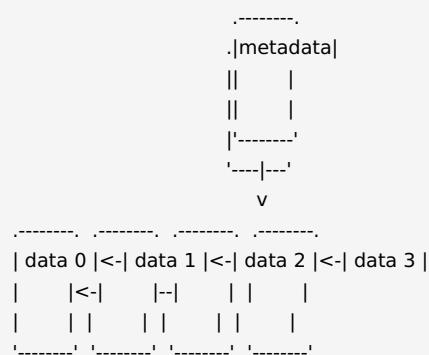
[flash translation layer \(FTL\)](#) to get a small power resilient filesystem with static wear leveling.

Files

Now that we have our building blocks out of the way, we can start looking at our filesystem as a whole.

The first step: How do we actually store our files?

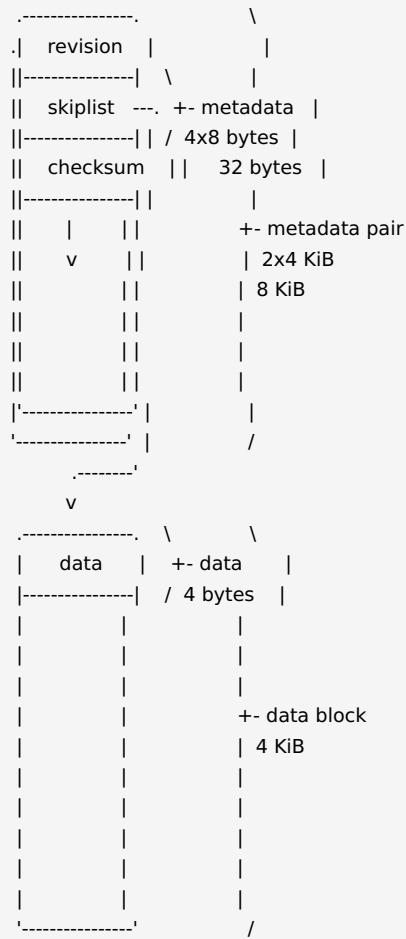
We've determined that CTZ skip-lists are pretty good at storing data compactly, so following the precedent found in other filesystems we could give each file a skip-list stored in a metadata pair that acts as an inode for the file.



However, this doesn't work well when files are small, which is common for embedded systems. Compared to PCs, *all* data in an embedded system is small.

Consider a small 4-byte file. With a two block metadata-pair and one block for the CTZ skip-list, we find ourselves using a full 3 blocks. On most NOR flash with 4 KiB blocks, this is 12 KiB of overhead. A ridiculous 3072x increase.

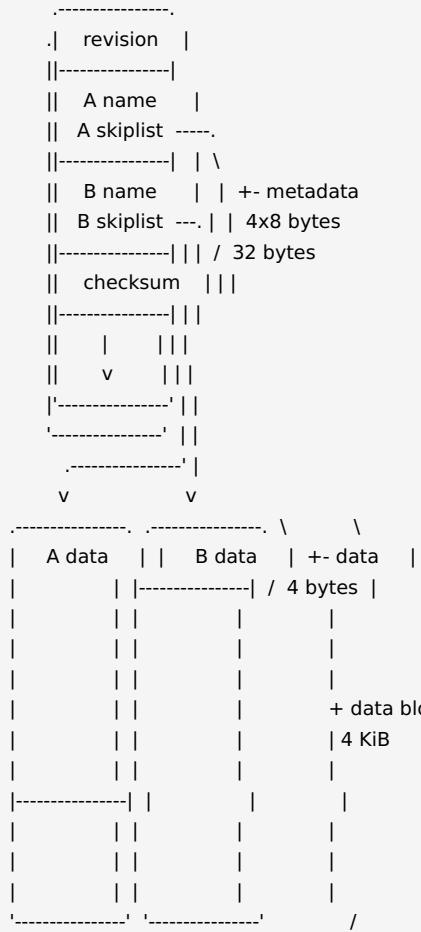
file stored as inode, 4 bytes costs ~12 KiB



We can make several improvements. First, instead of giving each file its own metadata pair, we can store multiple files in a single metadata pair. One way to do this is to directly associate a directory with a metadata pair (or a linked list of metadata pairs). This makes it easy for multiple files to share the directory's metadata pair for logging and reduces the collective storage overhead.

The strict binding of metadata pairs and directories also gives users direct control over storage utilization depending on how they organize their directories.

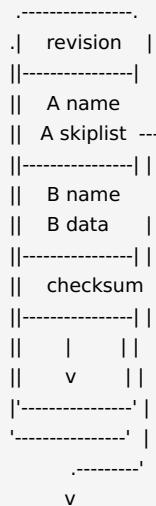
multiple files stored in metadata pair, 4 bytes costs ~4 KiB



The second improvement we can make is noticing that for very small files, our attempts to use CTZ skip-lists for compact storage backfires. Metadata pairs have a ~4x storage cost, so if our file is smaller than 1/4 the block size, there's actually no benefit in storing our file outside of our metadata pair.

In this case, we can store the file directly in our directory's metadata pair. We call this an inline file, and it allows a directory to store many small files quite efficiently. Our previous 4 byte file now only takes up a theoretical 16 bytes on disk.

inline files stored in metadata pair, 4 bytes costs ~16 bytes

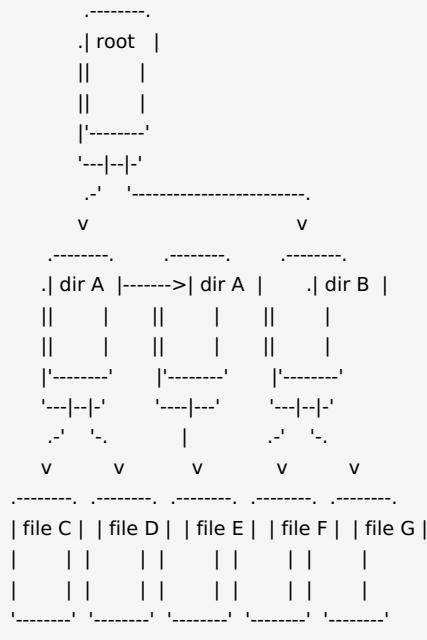


Once the file exceeds 1/4 the block size, we switch to a CTZ skip-list. This means that our files never use more than 4x storage overhead, decreasing as the file grows in size.

Directories

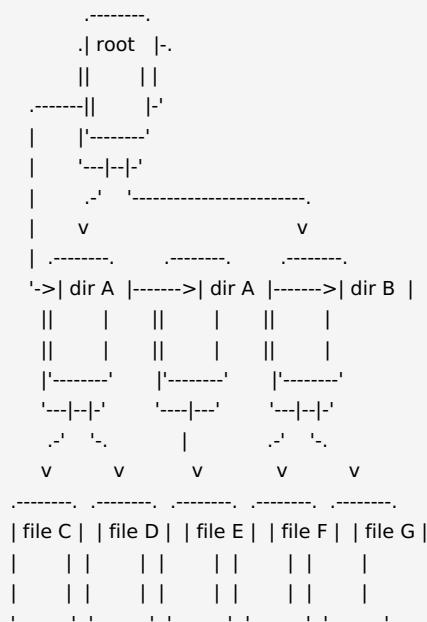
Now we just need directories to store our files. As mentioned above we want a strict binding of directories and metadata pairs, but there are a few complications we need to sort out.

On their own, each directory is a linked-list of metadata pairs. This lets us store an unlimited number of files in each directory, and we don't need to worry about the runtime complexity of unbounded logs. We can store other directory pointers in our metadata pairs, which gives us a directory tree, much like what you find on other filesystems.



The main complication is, once again, traversal with a constant amount of [RAM](#). The directory tree is a tree, and the unfortunate fact is you can't traverse a tree with constant RAM.

Fortunately, the elements of our tree are metadata pairs, so unlike CTZ skip-lists, we're not limited to strict COW operations. One thing we can do is thread a linked-list through our tree, explicitly enabling cheap traversal over the entire filesystem.



Unfortunately, not sticking to pure COW operations creates some problems. Now, whenever we want to manipulate the directory tree, multiple pointers need to be updated. If you're familiar with designing atomic data structures this should set off a bunch of red flags.

To work around this, our threaded linked-list has a bit of leeway. Instead of only containing metadata pairs found in our filesystem, it is allowed to contain metadata pairs that have no parent because of a power loss. These are called orphaned metadata pairs.

With the possibility of orphans, we can build power loss resilient operations that maintain a filesystem tree threaded with a linked-list for traversal.

Adding a directory to our tree:

```
-----.
.| root |-
||   ||
-----||   |-
|   '|-----'
|   '--|---|
|   .-'  '-.
|   v      v
| -----. -----
'->| dir A |->| dir C |
||   ||   ||
||   ||   ||
'|-----'|-----'
'-----' '-----'
```

allocate dir B

=>

```
-----.
.| root |-
||   ||
-----||   |-
|   '|-----'
|   '--|---|
|   .-'  '-.
|   v      v
| -----. -----
'->| dir A |--->| dir C |
||   | .->  ||
||   || ||   ||
'|-----'|-----'
'-----' |-----'
          |
-----| |
.| dir B |-
||   |
||   |
'|-----'
'-----'
```

insert dir B into threaded linked-list, creating an orphan

=>

```
-----.
.| root |-
||   ||
-----||   |-
|   '|-----'
|   '--|---|
|   .-'  '-----.
|   v      v
| -----. -----
'->| dir A |->| dir B |->| dir C |
||   || || orphan!||   ||
||   ||   || ||   ||
'|-----'|-----'|-----'
'-----' '-----' '-----'
```

add dir B to parent directory

=>

```
-----.
.| root |-
||   ||
-----||   |-
|   '|-----'
|   '--|---|
|   .-'  '-----.
|   v      v      v
| -----. -----
'->| dir A |->| dir B |->| dir C |
||   ||   || ||   ||
-----'-----'
```

```
||   ||   ||   |
|-----' |-----' |-----'
`-----' `-----'
```

Removing a directory:

```
-----.
.| root |-
||   ||
-----||   |'
|   |'-----'
|   '--|-|-'
|   .----' | '-----.
|   v     v     v
| .-----. .-----. .-----.
->| dir A |->| dir B |->| dir C |
||   ||   ||   ||
||   ||   ||   ||
|'-----' |'-----' |'-----'
`-----' `-----' `-----'
```

remove dir B from parent directory, creating an orphan

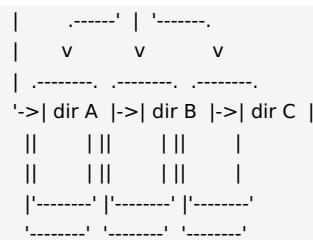
```
=>
-----.
.| root |-
||   ||
-----||   |'
|   |'-----'
|   '--|-|-'
|   .----' .-----.
|   v           v
| .-----. .-----. .-----.
->| dir A |->| dir B |->| dir C |
||   || orphan!! ||   ||
||   ||   ||   ||
|'-----' |'-----' |'-----'
`-----' `-----' `-----'
```

remove dir B from threaded linked-list, returning dir B to free blocks

```
=>
-----.
.| root |-
||   ||
-----||   |'
|   |'-----'
|   '--|-|-'
|   .----' .
|   v     v
| .-----.
->| dir A |->| dir C |
||   ||   ||
||   ||   ||
|'-----' |'-----'
`-----'
```

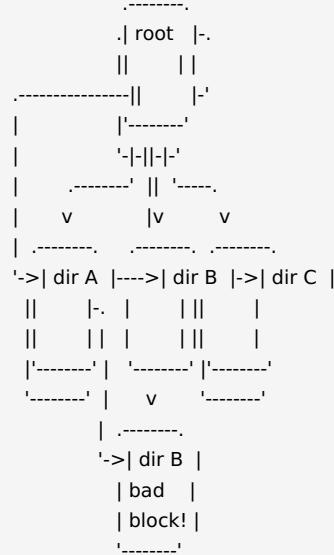
In addition to normal directory tree operations, we can use orphans to evict blocks in a metadata pair when the block goes bad or exceeds its allocated erases. If we lose power while evicting a metadata block we may end up with a situation where the filesystem references the replacement block while the threaded linked-list still contains the evicted block. We call this a half-orphan.

```
-----.
.| root |-
||   ||
-----||   |'
|   |'-----'
|   '--|-|-'
```



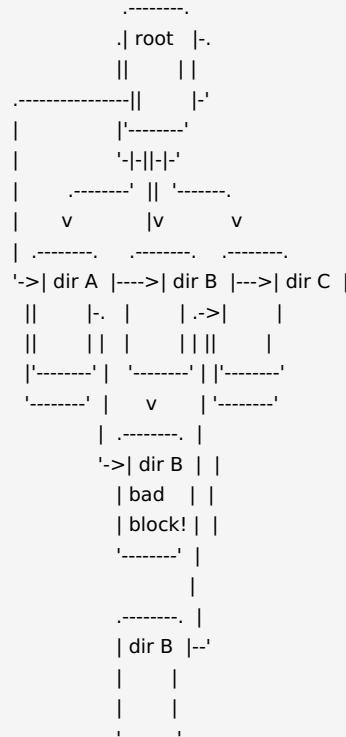
try to write to dir B

=>



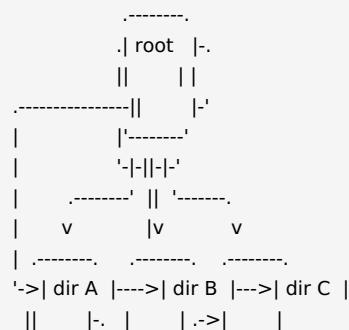
oh no! bad block detected, allocate replacement

=>



insert replacement in threaded linked-list, creating a half-orphan

=>



```

||   || |   || ||
'-----' | '-----' || '-----'
'-----' |   v   | '-----'
| .----- |
| | dir B | |
| | bad   | |
| | block! | |
| '-----' |
|   |
| .----- |
'->| dir B |--'
| half  |
| orphan!|
'-----'

```

fix reference in parent directory

=>

```

-----.
.| root |-
||   ||
-----||   |-
|   |'-----'
|   '|--|-|-'
|   .-----| '-----.
|   v     v     v
| .----- .----- .-----.
'->| dir A |->| dir B |->| dir C |
||   || |   ||   |
||   || |   ||   |
'-----' |'-----' |'-----'
'-----' |'-----' |'-----'

```

Finding orphans and half-orphans is expensive, requiring a $O(n^2)$ comparison of every metadata pair with every directory entry. But the tradeoff is a power resilient filesystem that works with only a bounded amount of RAM. Fortunately, we only need to check for orphans on the first allocation after boot, and a read-only littlefs can ignore the threaded linked-list entirely.

If we only had some sort of global state, then we could also store a flag and avoid searching for orphans unless we knew we were specifically interrupted while manipulating the directory tree (foreshadowing!).

The move problem

We have one last challenge: the move problem. Phrasing the problem is simple:

How do you atomically move a file between two directories?

In littlefs we can atomically commit to directories, but we can't create an atomic commit that spans multiple directories. The filesystem must go through a minimum of two distinct states to complete a move.

To make matters worse, file moves are a common form of synchronization for filesystems. As a filesystem designed for power-loss, it's important we get atomic moves right.

So what can we do?

- We definitely can't just let power-loss result in duplicated or lost files. This could easily break users' code and would only reveal itself in extreme cases. We were only able to be lazy about the threaded linked-list because it isn't user facing and we can handle the corner cases internally.
- Some filesystems propagate COW operations up the tree until a common parent

is found. Unfortunately this interacts poorly with our threaded tree and brings back the issue of upward propagation of wear.

- In a previous version of littlefs we tried to solve this problem by going back and forth between the source and destination, marking and unmarking the file as moving in order to make the move atomic from the user perspective. This worked, but not well. Finding failed moves was expensive and required a unique identifier for each file.

In the end, solving the move problem required creating a new mechanism for sharing knowledge between multiple metadata pairs. In littlefs this led to the introduction of a mechanism called "global state".

Global state is a small set of state that can be updated from *any* metadata pair. Combining global state with metadata pairs' ability to update multiple entries in one commit gives us a powerful tool for crafting complex atomic operations.

How does global state work?

Global state exists as a set of deltas that are distributed across the metadata pairs in the filesystem. The actual global state can be built out of these deltas by xor-ing together all of the deltas in the filesystem.

```
..... .-----.
.| |->| gdelta |->| |->| gdelta |->| gdelta |
|| | || 0x23 | || | || 0xff | || 0xce |  
|| | || | || | || | || |  
'-----' |-----' |-----' |-----' |-----'  
'-----' |-----' |-----' |-----' |-----'  
      |-----' |-----' |-----' |-----'  
      v       v       v  
0x00 --> xor -----> xor -----> xor --> gstate 0x12
```

To update the global state from a metadata pair, we take the global state we know and xor it with both our changes and any existing delta in the metadata pair. Committing this new delta to the metadata pair commits the changes to the filesystem's global state.

```
..... .-----.
.| |->| gdelta |->| |->| gdelta |->| gdelta |
|| | || 0x23 | || | || 0xff | || 0xce |  
|| | || | || | || | || |  
'-----' |-----' |-----' |-----' |-----'  
'-----' |-----' |-----' |-----' |-----'  
      v       v       v  
0x00 --> xor -----> xor -|-----> xor --> gstate = 0x12  
      |           |  
      |           |  
change gstate to 0xab --> xor <-----|-----'  
=>          |       v  
          '-----> xor  
          |  
          v  
..... .-----.
.| |->| gdelta |->| |->| gdelta |->| gdelta |
|| | || 0x23 | || | || 0x46 | || 0xce |  
|| | || | || | || | || |  
'-----' |-----' |-----' |-----' |-----'  
'-----' |-----' |-----' |-----' |-----'  
      v       v       v  
0x00 --> xor -----> xor -----> xor --> gstate = 0xab
```

To make this efficient, we always keep a copy of the global state in RAM. We only need to iterate over our metadata pairs and build the global state when the filesystem is mounted.

You may have noticed that global state is very expensive. We keep a copy in RAM and a delta in an unbounded number of metadata pairs. Even if we reset the global state to its initial value, we can't easily clean up the deltas on disk. For this reason, it's very important that we keep the size of global state bounded and extremely small. But, even with a strict budget, global state is incredibly valuable.

Now we can solve the move problem. We can create global state describing our move atomically with the creation of the new file, and we can clear this move state atomically with the removal of the old file.

```

----- gstate = no move
.| root |-
||   ||
-----||   |-
|   '|-----'
|   '-|-|-'
|   .----' | '----.
|   v     v     v
| .-----. ,-----. ,----.
'->| dir A |->| dir B |->| dir C |
||   ||   ||   ||   |
||   ||   ||   ||   |
'|-----' '|-----' '|-----'
'---|---' '---|---' '---|---'
      v
.----.
| file D |
|   |
|   |
'----.

```

begin move, add reference in dir C, change gstate to have move

=>

```

----- gstate = moving file D in dir A (m1)
.| root |-
||   ||
-----||   |-
|   '|-----'
|   '-|-|-'
|   .----' | '----.
|   v     v     v
| .-----. ,-----. ,----.
'->| dir A |->| dir B |->| dir C |
||   ||   ||   || gdelta |
||   ||   ||   ||=m1   |
'|-----' '|-----' '|-----'
'---|---' '---|---' '---|---'
      |
      v   v
.----.
| file D |
|   |
|   |
'----.

```

complete move, remove reference in dir A, change gstate to no move

=>

```

----- gstate = no move (m1^~m1)
.| root |-
||   ||
-----||   |-
|   '|-----'
|   '-|-|-'
|   .----' | '----.
|   v     v     v
| .-----. ,-----. ,----.
'->| dir A |->| dir B |->| dir C |
|| gdelta ||   || gdelta |
|| =~m1  ||   || =m1   |
'|-----' '|-----' '|-----'
'---|---' '---|---' '---|---'
      v
.----.
| file D |
|   |
|   |
'----.

```

If, after building our global state during mount, we find information describing an ongoing move, we know we lost power during a move and the file

is duplicated in both the source and destination directories. If this happens, we can resolve the move using the information in the global state to remove one of the files.

```
..... gstate = moving file D in dir A (m1)
.| root |-. ^
|| |-----> xor
|-----|| |-' ^ 
| |-----' | | | | | |
| |-----' |-----' | |
| | | | | | | |
| | .-----> xor -----> xor
| | v | v ^ v ^
| .-----| .-----| .-----|
'->| dir A |->| dir B |->| dir C ||
|| |-' || |-' || gdelta |-
|| | | | | | | =m1 |
|-----' |-----' |-----'
`---|-' |-----' |-----'
 | .-----'
 v v
.
| file D |
| |
| |
`-----'
```

We can also move directories the same way we move files. There is the threaded linked-list to consider, but leaving the threaded linked-list unchanged works fine as the order doesn't really matter.

----- gstate = no move ($m1 \sim m1$)

```

    .| root |.
    ||   ||
    -----||   |'
    |   |-----'
    |   '--|-|-'
    |   .-----' | '-----.
    |   v       v       v
    | .-----. .-----. .-----.
    '->| dir A |->| dir B |->| dir C |
    || gdelta ||      || gdelta |
    || =~m1  ||      || =m1   |
    '|-----' '|-----' '|-----'
    '-----' '-----' '-----|---'
                                v
    .-----.
    | file D |
    |   |
    |   |
    '-----'
  
```

begin move, add reference in dir C, change gstate to have move

```

=>
      .-----.
      | gstate = moving dir B in root (m1^~m1^m2)
      | | root |-
      | | | | | |
      |-----|| |-'|
      | |-----'|
      | | '--|-|-|-'|
      | | .-----' | '-----.
      | | v | v |
      | | .-----. | .-----.
      '->| dir A |-. | .->| dir C |
      || gdelta || | | || gdelta |
      || =~m1 || | | || =m1^m2 |
      '|-----' | | | || '|-----'
      '|-----' | | | || '--|-|-|-'|
      | | | .-----' |
      | | v v | v |
      | | .-----. | .-----.
      '->| dir B |-' | file D |
      || | | |
      || | | |
      || '|-----' | '-----'
      ||-----'

```

complete move, remove reference in root, change gstate to no move

=> .-----.
 gstate = no move (m1^~m1^m2^~m2)
 .| root |-.
 || gdelta ||
 .-----|| =~m2 |-'
 | |-----'
 | |---|-'
 | .----' '----.
 | v v
 | .-----. .-----.
 '->| dir A |-. .->| dir C |
 || gdelta || | || gdelta |
 || =~m1 || |-'|| =~m1^m2 |-----.
 '|-----'| |-----'|
 '|-----'| |-----'|
 | | .-'-' | |
 | | v v | |
 | .-----. .-----.|
 '->| dir B |--| file D |-'
 || | | | |
 || | | | |
 || .-----. '-----'

Global state gives us a powerful tool we can use to solve the move problem. And the result is surprisingly performant, only needing the minimum number of states and using the same number of commits as a naive move. Additionally, global state gives us a bit of persistent state we can use for some other small improvements.

Conclusion

And that's littlefs, thanks for reading!

Copyright (c) 2017, Arm Limited. All rights reserved.

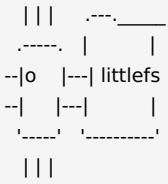
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of ARM nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

littlefs

A little fail-safe filesystem designed for microcontrollers.



Power-loss resilience - littlefs is designed to handle random power failures. All file operations have strong copy-on-write guarantees and if power is lost the filesystem will fall back to the last known good state.

Dynamic wear leveling - littlefs is designed with flash in mind, and provides wear leveling over dynamic blocks. Additionally, littlefs can detect bad blocks and work around them.

Bounded RAM/ROM - littlefs is designed to work with a small amount of memory. RAM usage is strictly bounded, which means RAM consumption does not change as the filesystem grows. The filesystem contains no unbounded

recursion and dynamic memory is limited to configurable buffers that can be provided statically.

Example

Here's a simple example that updates a file named `boot_count` every time main runs. The program can be interrupted at any time without losing track of how many times it has been booted and without corrupting the filesystem:

```
#include "lfs.h"

// variables used by the filesystem
lfs_t lfs;
lfs_file_t file;

// configuration of the filesystem is provided by this struct
const struct lfs_config cfg = {
    // block device operations
    .read   = user_provided_block_device_read,
    .prog   = user_provided_block_device_prog,
    .erase  = user_provided_block_device_erase,
    .sync   = user_provided_block_device_sync,

    // block device configuration
    .read_size = 16,
    .prog_size = 16,
    .block_size = 4096,
    .block_count = 128,
    .cache_size = 16,
    .lookahead_size = 16,
    .block_cycles = 500,
};

// entry point
int main(void) {
    // mount the filesystem
    int err = lfs_mount(&lfs, &cfg);

    // reformat if we can't mount the filesystem
    // this should only happen on the first boot
    if (err) {
        lfs_format(&lfs, &cfg);
        lfs_mount(&lfs, &cfg);
    }

    // read current count
    uint32_t boot_count = 0;
    lfs_file_open(&lfs, &file, "boot_count", LFS_O_RDWR | LFS_O_CREAT);
    lfs_file_read(&lfs, &file, &boot_count, sizeof(boot_count));

    // update boot count
    boot_count += 1;
    lfs_file_rewind(&lfs, &file);
    lfs_file_write(&lfs, &file, &boot_count, sizeof(boot_count));

    // remember the storage is not updated until the file is closed successfully
    lfs_file_close(&lfs, &file);

    // release any resources we were using
    lfs_unmount(&lfs);

    // print the boot count
    printf("boot_count: %d\n", boot_count);
}
```

Usage

Detailed documentation (or at least as much detail as is currently available) can be found in the comments in [lfs.h](#).

littlefs takes in a configuration structure that defines how the filesystem operates. The configuration struct provides the filesystem with the block device operations and dimensions, tweakable parameters that tradeoff memory usage for performance, and optional static buffers if the user wants to avoid dynamic memory.

The state of the littlefs is stored in the `lfs_t` type which is left up to the user to allocate, allowing multiple filesystems to be in use simultaneously. With the `lfs_t` and configuration struct, a user can format a block device or mount the filesystem.

Once mounted, the littlefs provides a full set of POSIX-like file and directory functions, with the deviation that the allocation of filesystem structures must be provided by the user.

All POSIX operations, such as remove and rename, are atomic, even in event of power-loss. Additionally, file updates are not actually committed to the filesystem until sync or close is called on the file.

Other notes

All littlefs calls have the potential to return a negative error code. The errors can be either one of those found in the enum `lfs_error` in [lfs.h](#), or an error returned by the user's block device operations.

In the configuration struct, the `prog` and `erase` function provided by the user may return a `LFS_ERR_CORRUPT` error if the implementation already can detect corrupt blocks. However, the wear leveling does not depend on the return code of these functions, instead all data is read back and checked for integrity.

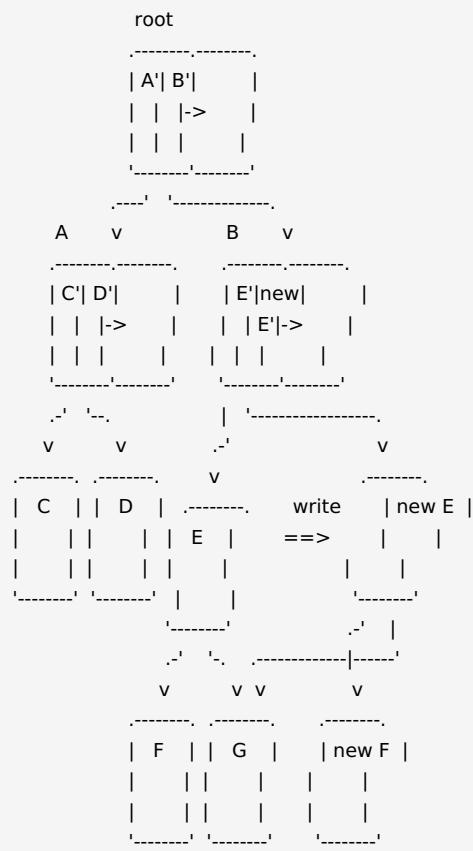
If your storage caches writes, make sure that the provided `sync` function flushes all the data to memory and ensures that the next read fetches the data from memory, otherwise data integrity can not be guaranteed. If the `write` function does not perform caching, and therefore each `read` or `write` call hits the memory, the `sync` function can simply return 0.

Design

At a high level, littlefs is a block based filesystem that uses small logs to store metadata and larger copy-on-write (COW) structures to store file data.

In littlefs, these ingredients form a sort of two-layered cake, with the small logs (called metadata pairs) providing fast updates to metadata anywhere on storage, while the COW structures store file data compactly and without any wear amplification cost.

Both of these data structures are built out of blocks, which are fed by a common block allocator. By limiting the number of erases allowed on a block per allocation, the allocator provides dynamic wear leveling over the entire filesystem.



More details on how littlefs works can be found in [DESIGN.md](#) and [SPEC.md](#).

- [DESIGN.md](#) - A fully detailed dive into how littlefs works.
I would suggest reading it as the tradeoffs at work are quite interesting.
- [SPEC.md](#) - The on-disk specification of littlefs with all the nitty-gritty details. May be useful for tooling development.

Testing

The littlefs comes with a test suite designed to run on a PC using the [emulated block device](#) found in the emubd directory.

The tests assume a Linux environment and can be started with make:

```
make test
```

License

The littlefs is provided under the [BSD-3-Clause](#) license. See [LICENSE.md](#) for more information. Contributions to this project are accepted under the same license.

Individual files contain the following tag instead of the full license text.

```
SPDX-License-Identifier: BSD-3-Clause
```

This enables machine processing of license information based on the SPDX License Identifiers that are here available: <http://spdx.org/licenses/>

Related projects

- [littlefs-fuse](#) - A FUSE wrapper for littlefs. The project allows you to mount littlefs directly on a Linux machine. Can be useful for debugging littlefs if you have an SD card handy.
- [littlefs-js](#) - A javascript wrapper for littlefs. I'm not sure why you would want this, but it is handy for demos. You can see it in action [here](#).
- [mkifs](#) - A command line tool built by the [Lua RTOS](#) guys for making littlefs images from a host PC. Supports Windows, Mac OS, and Linux.
- [Mbed OS](#) - The easiest way to get started with littlefs is to jump into Mbed which already has block device drivers for most forms of embedded storage. littlefs is available in Mbed OS as the [LittleFileSystem](#) class.
- [SPIFFS](#) - Another excellent embedded filesystem for NOR flash. As a more traditional logging filesystem with full static wear-leveling, SPIFFS will likely outperform littlefs on small memories such as the internal flash on microcontrollers.
- [Dhara](#) - An interesting NAND flash translation layer designed for small MCUs. It offers static wear-leveling and power-resilience with only a fixed $O(/address/)$ pointer structure stored on each block and in RAM.

littlefs technical specification

This is the technical specification of the little filesystem. This document covers the technical details of how the littlefs is stored on disk for introspection and tooling. This document assumes you are familiar with the design of the littlefs, for more info on how littlefs works check out [DESIGN.md](#).

```
  |||  .-._____
  .----. |   |
--|o  |---| littlefs |
--|_ |---|   |
'___' '-----'
  |||
```

Some quick notes

- littlefs is a block-based filesystem. The disk is divided into an array of evenly sized blocks that are used as the logical unit of storage.
- Block pointers are stored in 32 bits, with the special value `0xffffffff` representing a null block address.
- In addition to the logical block size (which usually matches the erase block size), littlefs also uses a program block size and read block size. These determine the alignment of block device operations, but don't need to be consistent for portability.
- By default, all values in littlefs are stored in little-endian byte order.

Directories / Metadata pairs

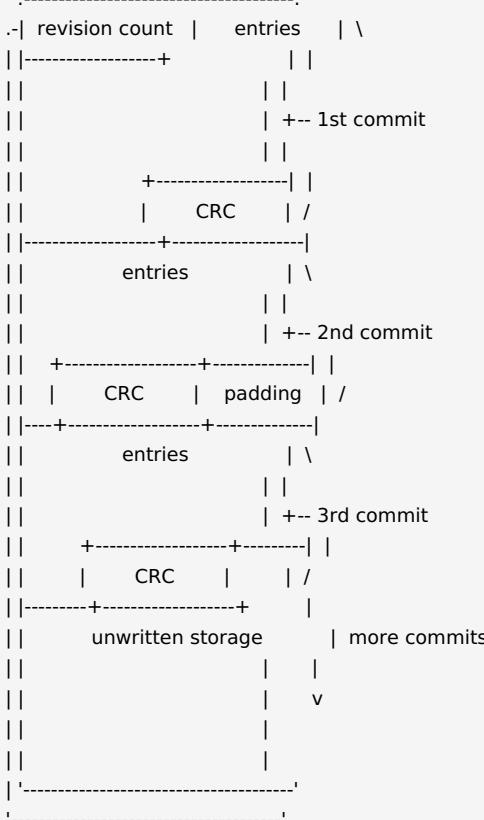
Metadata pairs form the backbone of littlefs and provide a system for distributed atomic updates. Even the superblock is stored in a metadata pair.

As their name suggests, a metadata pair is stored in two blocks, with one block providing a backup during erase cycles in case power is lost. These two blocks are not necessarily sequential and may be anywhere on disk, so a "pointer" to a

metadata pair is stored as two block pointers.

On top of this, each metadata block behaves as an appendable log, containing a variable number of commits. Commits can be appended to the metadata log in order to update the metadata without requiring an erase cycles. Note that successive commits may supersede the metadata in previous commits. Only the most recent metadata should be considered valid.

The high-level layout of a metadata block is fairly simple:



Each metadata block contains a 32-bit revision count followed by a number of commits. Each commit contains a variable number of metadata entries followed by a 32-bit CRC.

Note also that entries aren't necessarily word-aligned. This allows us to store metadata more compactly, however we can only write to addresses that are aligned to our program block size. This means each commit may have padding for alignment.

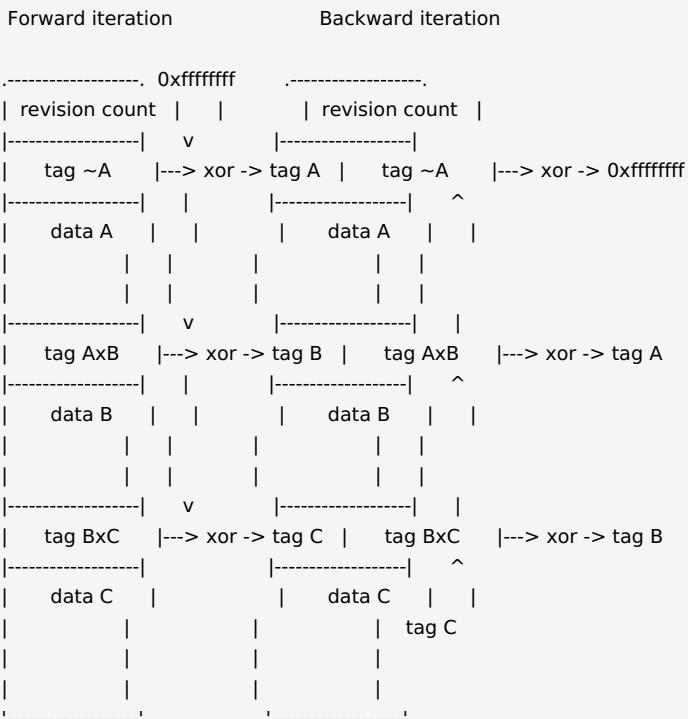
Metadata block fields:

1. **Revision count (32-bits)** - Incremented every erase cycle. If both blocks contain valid commits, only the block with the most recent revision count should be used. Sequence comparison must be used to avoid issues with integer overflow.
2. **CRC (32-bits)** - Detects corruption from power-loss or other write issues. Uses a CRC-32 with a polynomial of `0x04c11db7` initialized with `0xffffffff`.

Entries themselves are stored as a 32-bit tag followed by a variable length blob of data. But exactly how these tags are stored is a little bit tricky.

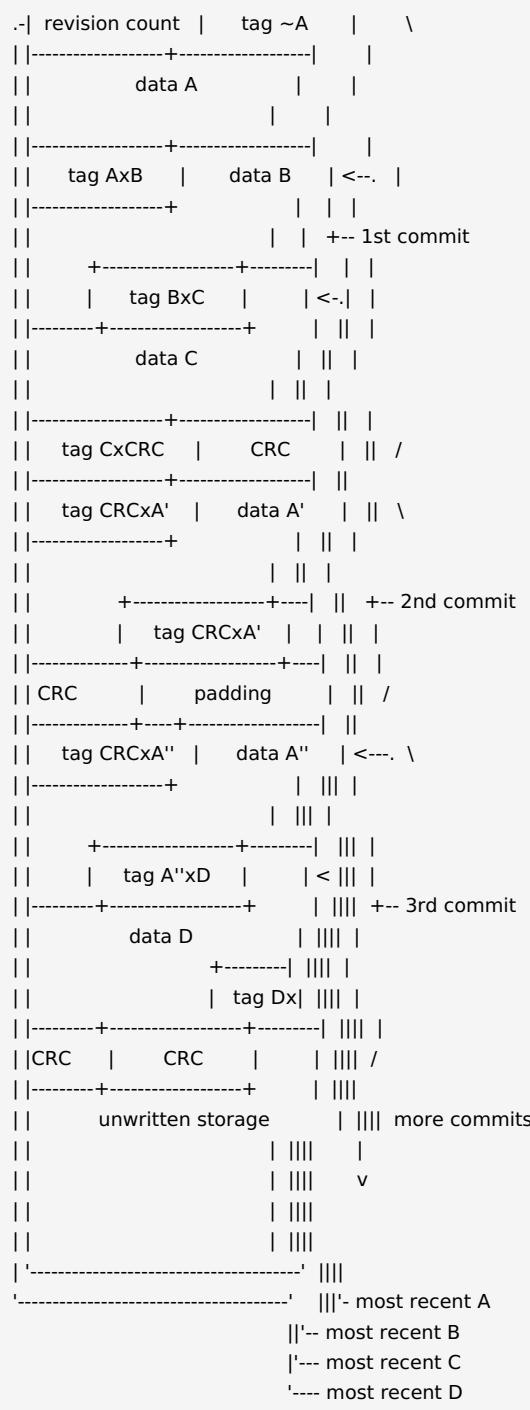
Metadata blocks support both forward and backward iteration. In order to do this without duplicating the space for each tag, neighboring entries have their

tags XORed together, starting with 0xffffffff.



One last thing to note before we get into the details around tag encoding. Each tag contains a valid bit used to indicate if the tag and containing commit is valid. This valid bit is the first bit found in the tag and the commit and can be used to tell if we've attempted to write to the remaining space in the block.

Here's a more complete example of metadata block containing 4 entries:



Metadata tags

So in littlefs, 32-bit tags describe every type of metadata. And this means every type of metadata, including file entries, directory fields, and global state. Even the CRCs used to mark the end of commits get their own tag.

Because of this, the tag format contains some densely packed information. Note that there are multiple levels of types which break down into more info:

```

[---      32      ---]
[1|-- 11  --|-- 10 --|-- 10 --]
^   ^   .   ^      ^- length
|.   |   .   '----- id
|.   '----- type (type3)
'-----,----- valid bit
[-3|- 8 --]
^   ^- chunk
'----- type (type1)

```

Before we go further, there's one important thing to note. These tags are **not** stored in little-endian. Tags stored in commits are actually stored in big-endian (and is the only thing in littlefs stored in big-endian). This little bit of craziness comes from the fact that the valid bit must be the first bit in a commit, and when converted to little-endian, the valid bit finds itself in byte 4. We could restructure the tag to store the valid bit lower, but, because none of the fields are byte-aligned, this would be more complicated than just storing the tag in big-endian.

Another thing to note is that both the tags `0x00000000` and `0xffffffff` are invalid and can be used for null values.

Metadata tag fields:

1. **Valid bit (1-bit)** - Indicates if the tag is valid.
2. **Type3 (11-bits)** - Type of the tag. This field is broken down further into a 3-bit abstract type and an 8-bit chunk field. Note that the value `0x000` is invalid and not assigned a type.
3. **Type1 (3-bits)** - Abstract type of the tag. Groups the tags into 8 categories that facilitate bitmasked lookups.
4. **Chunk (8-bits)** - Chunk field used for various purposes by the different abstract types. `type1+chunk+id` form a unique identifier for each tag in the metadata block.
5. **Id (10-bits)** - File id associated with the tag. Each file in a metadata block gets a unique id which is used to associate tags with that file. The special value `0x3ff` is used for any tags that are not associated with a file, such as directory and global metadata.
6. **Length (10-bits)** - Length of the data in bytes. The special value `0x3ff` indicates that this tag has been deleted.

Metadata types

What follows is an exhaustive list of metadata in littlefs.

0x401 LFS_TYPE_CREATE

Creates a new file with this id. Note that files in a metadata block don't necessarily need a create tag. All a create does is move over any files using this id. In this sense a create is similar to insertion into an imaginary array of files.

The create and delete tags allow littlefs to keep files in a directory ordered alphabetically by filename.

0x4ff LFS_TYPE_DELETE

Deletes the file with this id. An inverse to create, this tag moves over any files neighboring this id similar to a deletion from an imaginary array of files.

0x0xx LFS_TYPE_NAME

Associates the id with a file name and file type.

The data contains the file name stored as an ASCII string (may be expanded to UTF8 in the future).

The chunk field in this tag indicates an 8-bit file type which can be one of the following.

Currently, the name tag must precede any other tags associated with the id and can not be reassigned without deleting the file.

Layout of the name tag:

```
tag          data
[-- 32 --][--- variable length ---]
[1| 3| 8 | 10 | 10 ][---      (size)      ---]
^ ^ ^ ^ - size      ^- file name
| | | '---- id
| | '----- file type
| '----- type1 (0x0)
'----- valid bit
```

Name fields:

1. **file type (8-bits)** - Type of the file.
 2. **file name** - File name stored as an ASCII string.
-

0x001 LFS_TYPE_REG

Initializes the id + name as a regular file.

How each file is stored depends on its struct tag, which is described below.

0x002 LFS_TYPE_DIR

Initializes the id + name as a directory.

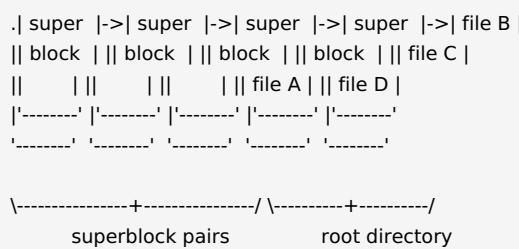
Directories in littlefs are stored on disk as a linked-list of metadata pairs, each pair containing any number of files in alphabetical order. A pointer to the directory is stored in the struct tag, which is described below.

0x0ff LFS_TYPE_SUPERBLOCK

Initializes the id as a superblock entry.

The superblock entry is a special entry used to store format-time configuration and identify the filesystem.

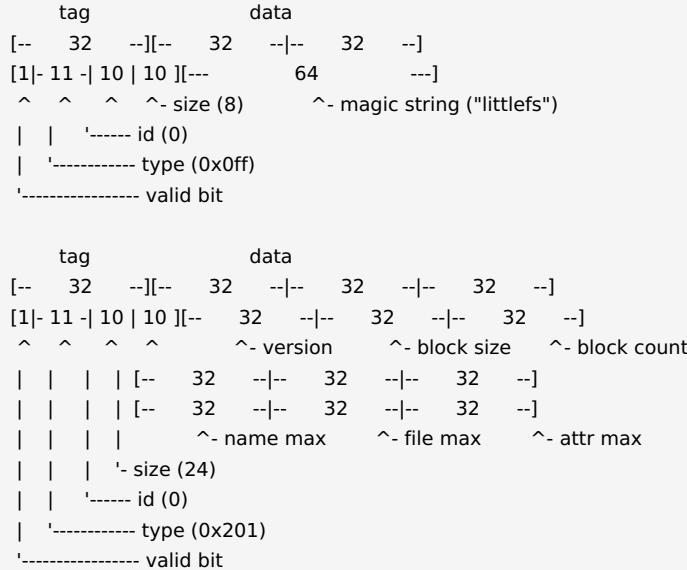
The name is a bit of a misnomer. While the superblock entry serves the same purpose as a superblock found in other filesystems, in littlefs the superblock does not get a dedicated block. Instead, the superblock entry is duplicated across a linked-list of metadata pairs rooted on the blocks 0 and 1. The last metadata pair doubles as the root directory of the filesystem.



The filesystem starts with only the root directory. The superblock metadata pairs grow every time the root pair is compacted in order to prolong the life of the device exponentially.

The contents of the superblock entry are stored in a name tag with the superblock type and an inline-struct tag. The name tag contains the magic string "littlefs", while the inline-struct tag contains version and configuration information.

Layout of the superblock name tag and inline-struct tag:



Superblock fields:

1. **Magic string (8-bytes)** - Magic string indicating the presence of littlefs on the device. Must be the string "littlefs".
2. **Version (32-bits)** - The version of littlefs at format time. The version is encoded in a 32-bit value with the upper 16-bits containing the major version, and the lower 16-bits containing the minor version.
This specification describes version 2.0 (0x00020000).
3. **Block size (32-bits)** - Size of the logical block size used by the filesystem in bytes.
4. **Block count (32-bits)** - Number of blocks in the filesystem.
5. **Name max (32-bits)** - Maximum size of file names in bytes.
6. **File max (32-bits)** - Maximum size of files in bytes.
7. **Attr max (32-bits)** - Maximum size of file attributes in bytes.

The superblock must always be the first entry (id 0) in a metadata pair as well as be the first entry written to the block. This means that the superblock entry can be read from a device using offsets alone.

0x2xx LFS_TYPE_STRUCT

Associates the id with an on-disk data structure.

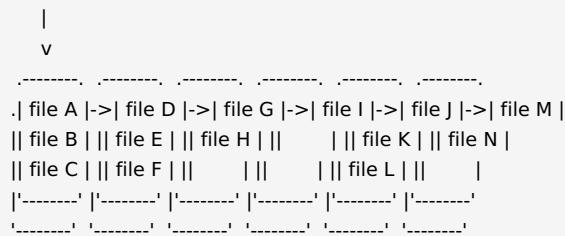
The exact layout of the data depends on the data structure type stored in the chunk field and can be one of the following.

Any type of struct supersedes all other structs associated with the id. For example, appending a ctz-struct replaces an inline-struct on the same file.

0x200 LFS_TYPE_DIRSTRUCT

Gives the id a directory data structure.

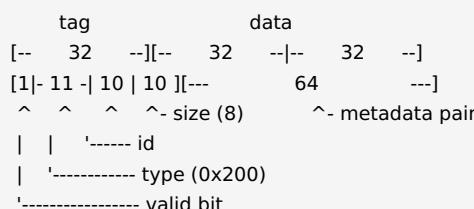
Directories in littlefs are stored on disk as a linked-list of metadata pairs, each pair containing any number of files in alphabetical order.



The dir-struct tag contains only the pointer to the first metadata-pair in the directory. The directory size is not known without traversing the directory.

The pointer to the next metadata-pair in the directory is stored in a tail tag, which is described below.

Layout of the dir-struct tag:



Dir-struct fields:

1. **Metadata pair (8-bytes)** - Pointer to the first metadata-pair in the directory.

0x201 LFS_TYPE_INLINESTRUCT

Gives the id an inline data structure.

Inline structs store small files that can fit in the metadata pair. In this case, the file data is stored directly in the tag's data area.

Layout of the inline-struct tag:

```

tag          data
[-- 32 --][-- variable length   --]
[1|- 11 -| 10 | 10 ][--      (size)      --]
^  ^  ^  ^- size      ^- inline data
|  | '---- id
| '----- type (0x201)
'----- valid bit

```

Inline-struct fields:

1. **Inline data** - File data stored directly in the metadata-pair.

0x202 LFS_TYPE_C TZSTRUCT

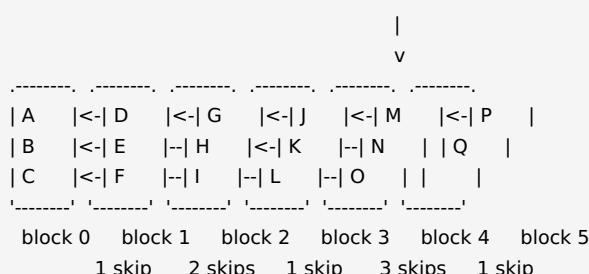
Gives the id a CTZ skip-list data structure.

CTZ skip-lists store files that can not fit in the metadata pair. These files are stored in a skip-list in reverse, with a pointer to the head of the skip-list. Note that the head of the skip-list and the file size is enough information to read the file.

How exactly CTZ skip-lists work is a bit complicated. A full explanation can be found in the [DESIGN.md](#).

A quick summary: For every n th block where n is divisible by 2^x , that block contains a pointer to block $n-2^x$.

These pointers are stored in increasing order of x in each block of the file before the actual data.



Note that the maximum number of pointers in a block is bounded by the maximum file size divided by the block size. With 32 bits for file size, this results in a minimum block size of 104 bytes.

Layout of the CTZ-struct tag:

```

tag          data
[-- 32 --][-- 32 --|-- 32 --]
[1|- 11 -| 10 | 10 ][-- 32 --|-- 32 --]
^  ^  ^  ^      ^- file size
|  |  |  |      '----- file head
|  |  | '--- size (8)
|  | '---- id
| '----- type (0x202)
'----- valid bit

```

CTZ-struct fields:

1. **File head (32-bits)** - Pointer to the block that is the head of the file's CTZ skip-list.
2. **File size (32-bits)** - Size of the file in bytes.

0x3xx LFS_TYPE_USERATTR

Attaches a user attribute to an id.

littlefs has a concept of "user attributes". These are small user-provided attributes that can be used to store things like timestamps, hashes, permissions, etc.

Each user attribute is uniquely identified by an 8-bit type which is stored in the chunk field, and the user attribute itself can be found in the tag's data.

There are currently no standard user attributes and a portable littlefs implementation should work with any user attributes missing.

Layout of the user-attr tag:

```
    tag          data
[-- 32  --][--- variable length   ---]
[1| 3| 8 | 10 | 10 ][---      (size)      ---]
^ ^ ^ ^ ^- size      ^- attr data
| | | '---- id
| | '----- attr type
| '----- type1 (0x3)
'----- valid bit
```

User-attr fields:

1. **Attr type (8-bits)** - Type of the user attributes.
2. **Attr data** - The data associated with the user attribute.

0x6xx LFS_TYPE_TAIL

Provides the tail pointer for the metadata pair itself.

The metadata pair's tail pointer is used in littlefs for a linked-list containing all metadata pairs. The chunk field contains the type of the tail, which indicates if the following metadata pair is a part of the directory (hard-tail) or only used to traverse the filesystem (soft-tail).

```
-----.
.| dir A |-
||softtail|
-----|     |'
|     '|-----'
|     '--|--|'
|     .-'  -----.
|     v       v
|     -----. -----. -----.
'->| dir B |->| dir B |->| dir C |
||hardtail||softtail||    |
||  ||  ||  ||  |
|-----'|-----'|-----'
|-----'|-----'|-----'
```

Currently any type supersedes any other preceding tails in the metadata pair, but this may change if additional metadata pair state is added.

A note about the metadata pair linked-list: Normally, this linked-list contains every metadata pair in the filesystem. However, there are some operations that

can cause this linked-list to become out of sync if a power-loss were to occur.

When this happens, littlefs sets the "sync" flag in the global state. How exactly this flag is stored is described below.

When the sync flag is set:

1. The linked-list may contain an orphaned directory that has been removed in the filesystem.
2. The linked-list may contain a metadata pair with a bad block that has been replaced in the filesystem.

If the sync flag is set, the threaded linked-list must be checked for these errors before it can be used reliably. Note that the threaded linked-list can be ignored if littlefs is mounted read-only.

Layout of the tail tag:

```
tag          data
[-- 32 --][-- 32 --|--- 32 --]
[1| 3| 8 | 10 | 10 ][---       64      ---]
^ ^ ^ ^ ^- size (8)      ^- metadata pair
| | | '---- id
| | '----- tail type
| '----- type1 (0x6)
'----- valid bit
```

Tail fields:

1. **Tail type (8-bits)** - Type of the tail pointer.
2. **Metadata pair (8-bytes)** - Pointer to the next metadata-pair.

0x600 LFS_TYPE_SOFTTAIL

Provides a tail pointer that points to the next metadata pair in the filesystem.

In this case, the next metadata pair is not a part of our current directory and should only be followed when traversing the entire filesystem.

0x601 LFS_TYPE_HARDDTAIL

Provides a tail pointer that points to the next metadata pair in the directory.

In this case, the next metadata pair belongs to the current directory. Note that because directories in littlefs are sorted alphabetically, the next metadata pair should only contain filenames greater than any filename in the current pair.

0x7xx LFS_TYPE_GSTATE

Provides delta bits for global state entries.

littlefs has a concept of "global state". This is a small set of state that can be updated by a commit to *any* metadata pair in the filesystem.

The way this works is that the global state is stored as a set of deltas distributed across the filesystem such that the global state can be found by the xor-sum of these deltas.

```
.....-----.
.| |->| gdelta |->| |->| gdelta |->| gdelta |
|| || 0x23 || || || 0xff || || 0xce ||
|| || || || || || || |
'-----' |-----' |-----' |-----' |-----'
'-----' |-----' |-----' |-----' |-----'
           v         v         v
0x00 --> xor -----> xor -----> xor --> gstate = 0x12
```

Note that storing globals this way is very expensive in terms of storage usage, so any global state should be kept very small.

The size and format of each piece of global state depends on the type, which is stored in the chunk field. Currently, the only global state is move state, which is outlined below.

0x7ff LFS_TYPE_MOVESTATE

Provides delta bits for the global move state.

The move state in littlefs is used to store info about operations that could cause to filesystem to go out of sync if the power is lost. The operations where this could occur is moves of files between metadata pairs and any operation that changes metadata pairs on the threaded linked-list.

In the case of moves, the move state contains a tag + metadata pair describing the source of the ongoing move. If this tag is non-zero, that means that power was lost during a move, and the file exists in two different locations. If this happens, the source of the move should be considered deleted, and the move should be completed (the source should be deleted) before any other write operations to the filesystem.

In the case of operations to the threaded linked-list, a single "sync" bit is used to indicate that a modification is ongoing. If this sync flag is set, the threaded linked-list will need to be checked for errors before it can be used reliably. The exact cases to check for are described above in the tail tag.

Layout of the move state:

```
tag                      data
[-- 32 --][-- 32 --|--- 32 --|--- 32 --]
[1|- 11 -| 10 | 10 ][1|- 11 -| 10 | 10 |--- 64 ---]
^ ^ ^ ^ ^ ^ ^ ^ padding (0) ^- metadata pair
| | | | | | '---- move id
| | | | | | '----- move type
| | | | | | '----- sync bit
| | | | |
| | | | '- size (12)
| | | | '---- id (0x3ff)
| | | | '----- type (0x7ff)
|----- valid bit
```

Move state fields:

1. **Sync bit (1-bit)** - Indicates if the metadata pair threaded linked-list is in-sync. If set, the threaded linked-list should be checked for errors.

2. **Move type (11-bits)** - Type of move being performed. Must be either `0x000`, indicating no move, or `0x4ff` indicating the source file should be deleted.
 3. **Move id (10-bits)** - The file id being moved.
 4. **Metadata pair (8-bytes)** - Pointer to the metadata-pair containing the move.
-

0x5xx LFS_TYPE_CRC

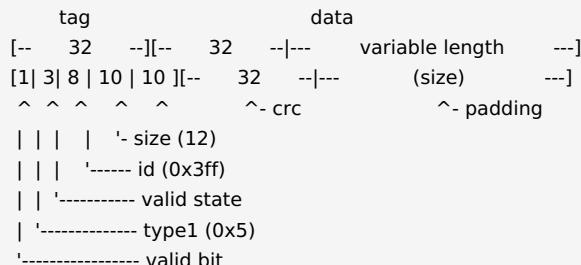
Last but not least, the CRC tag marks the end of a commit and provides a checksum for any commits to the metadata block.

The first 32-bits of the data contain a CRC-32 with a polynomial of `0x04c11db7` initialized with `0xffffffff`. This CRC provides a checksum for all metadata since the previous CRC tag, including the CRC tag itself. For the first commit, this includes the revision count for the metadata block.

However, the size of the data is not limited to 32-bits. The data field may larger to pad the commit to the next program-aligned boundary.

In addition, the CRC tag's chunk field contains a set of flags which can change the behaviour of commits. Currently the only flag in use is the lowest bit, which determines the expected state of the valid bit for any following tags. This is used to guarantee that unwritten storage in a metadata block will be detected as invalid.

Layout of the CRC tag:



CRC fields:

1. **Valid state (1-bit)** - Indicates the expected value of the valid bit for any tags in the next commit.
 2. **CRC (32-bits)** - CRC-32 with a polynomial of `0x04c11db7` initialized with `0xffffffff`.
 3. **Padding** - Padding to the next program-aligned boundary. No guarantees are made about the contents.
-

Mongoose OS support for littlefs

<https://github.com/ARMmbed/littlefs>

Mongoose OS library for MQ135 GAS SENSOR

Getting Started

1. Download and install [Mongoose OS](#).
2. You have a complete [documentation](#) and [video](#) tutorials about Mongoose OS.

Usage of MQ135 Library

Any GPIO pin with ADC capability can be used

```
MQ.MQ135.attach(pin);
```

You can use ambient temperature/humidity correction factor:

```
MQ.MQ135.getCorrectedPPM(temperature, humidity);
```

Note:

- This type of sensor needs 24h after powered to "warning-up" (calibration).

Example of MQ135 Library

See [example](#).

Acknowledgements

Many thanks to the developer of the mgos_stmpe610 library as the
this library uses the same structure.

Many thanks to the Mongoose OS developers

ADS7843

Mongoose native SPI driver for ADS7843/XPT2046 Touch Screen Controller

Introduction

ADS7843/XPT2046 is a chip that is connected to a resistive touchscreen.

The touchscreen has resistance in the X and Y directions separately.

When the user touches the touchscreen a resistance in both directions
is detected by the touchscreen controller.

The touchscreen controller records the X and Y resistance changes using
an ADC that can measure the voltage from the resistance changes in the
X and Y directions. A schematic of the test board used is included in
this project.

API

This driver initializes the chip and allows the user to register a callback,

which will be called each time the driver senses a touch (TOUCH_DOWN) or a release (TOUCH_UP) event. The callback function will be given the X and Y coordinates where the user pressed the screen. The X and Y coordinates are passed in the mgos_ads7843_event_data structure. The x and y attributes are values in screen pixels.

Details

The SPI bus must be connected to the screen via 5 signals. These are MOSI, MISO, SCK, /CS and /IRQ. When the screen is touched by the user the /IRQ line goes low. In the microcontroller code this causes an interrupt to occur. The interrupt service routine then reads the X and Y ADC values from the ADS7843 device (using the MOSI, MISO, SCK, /CS signals).

The interrupt service routine then converts these ADC values into screen pixel positions. Once this is done a callback function is called. A pointer to a structure that holds the above X and y values along with some other data is passed to the callback function.

Example Application

mos.yml

The driver uses the Mongoose native SPI driver. It is configured by setting up the MOSI, MISO, SCLK pins and assigning one of the three available CS positions, in this example we use CS1:

```
config_schema:
- ["spi.enable", true]
- ["spi.mosi_gpio", 23]
- ["spi.miso_gpio", 19]
- ["spi.sclk_gpio", 18]
- ["spi.cs1_gpio", 27 ] # This defines the ADS7843 SPI chip select pin
- ["ads7843", "o", {title: "ADS7843/XPT2046 TouchScreen"}]
- ["ads7843.cs_index", "i", 1, {title: "spi.cs*_gpio index, 0, 1 or 2"}] # This defines the SPI CS line to use (0, 1 or 2)
- ["ads7843.irq_pin", "i", 25, {title: "IRQ pin (taken low when the display is touched.)"}]
- ["ads7843.x_pixels", "i", 320, {title: "The display pixel count in the horizontal direction"}]
- ["ads7843.y_pixels", "i", 240, {title: "The display pixel count in the vertical direction"}]
- ["ads7843.flip_x", "i", 0, {title: "Flip the X direction (0/1, 0 = no flip). Use this if the x direction is reversed."}]
- ["ads7843.flip_y", "i", 0, {title: "Flip the Y direction (0/1, 0 = no flip). Use this if the y direction is reversed."}]
- ["ads7843.flip_x_y", "i", 0, {title: "Flip the X and Y directions (0/1, 0 = no flip). Use this if the display is upside down."}]
- ["ads7843.min_x_adc", "i", 12, {title: "The min X axis ADC calibration value. Enter the value from debug output (min adc x value at screen edge)."}]
- ["ads7843.max_x_adc", "i", 121, {title: "The max X axis ADC calibration value. Enter the value from debug output (max adc x value at screen edge)."}]
- ["ads7843.min_y_adc", "i", 7, {title: "The min Y axis ADC calibration value. Enter the value from debug output (min adc y value at screen edge)."}]
- ["ads7843.max_y_adc", "i", 118, {title: "The max Y axis ADC calibration value. Enter the value from debug output (max adc y value at screen edge)."}]
```

Application

```

#include "mgos.h"
#include "mgos_ads7843.h"

static void touch_handler(struct mgos_ads7843_event_data *event_data) {
    if (!event_data) {
        return;
    }

    LOG(LL_INFO, ("orientation=%s", event_data->orientation ? "PORTRAIT" : "LANDSCAPE"));
    LOG(LL_INFO, ("Touch %s, down for %.1f seconds", event_data->direction==TOUCH_UP?"UP":"DOWN", event_data->down_seconds));
    LOG(LL_INFO, ("pixels x/y = %d/%d, adc x/y = %d/%d", event_data->x, event_data->y, event_data->x_adc, event_data->y_adc));

}

enum mgos_app_init_result mgos_app_init(void) {

    mgos_ads7843_set_handler(touch_handler);

    return MGOS_APP_INIT_SUCCESS;
}

```

Screen calibration

The min_x_adc, max_x_adc, min_y_adc and max_y_adc should be set in the mos.yml file in order to calibrate the display. This allows the pixel positions returned to accurately represent the position the display was touched.

In order to calibrate the display run the example program and touch each edge of the display.

When running the example application debug data is sent out on the serial port when the display is touched.

E.G

```
[Jan 9 02:36:10.761] touch_handler      orientation=PORTRAIT
[Jan 9 02:36:10.766] touch_handler      Touch DOWN, down for 0.0 seconds
[Jan 9 02:36:10.772] touch_handler      pixels x/y = 0/4, adc x/y = 10/9
```

The above shows the top left corner of the display being touched. The 'adc x/y = 10/9' text contains the min X and Y values. The min_x_adc and min_y_adc attributes in the example mos.yml should be set to the values found.

This should be repeated for the bottom right corner of the display using the max_x_adc and max_y_adc values.

E.G

```
[Jan 9 02:36:15.110] touch_handler      orientation=PORTRAIT
[Jan 9 02:36:15.115] touch_handler      Touch DOWN, down for 0.3 seconds
[Jan 9 02:36:15.120] touch_handler      pixels x/y = 317/233, adc x/y = 120/115
```

Testing

This driver has been tested with the following display

http://www.lcdwiki.com/2.8inch_SPI_Module_ILI9341_SKU:MSP2807

in all four orientations using a modified version of the following project

<https://github.com/mongoose-os-apps/huzzah-featherwing>

Google IoT Core integration for Mongoose OS

This library implements integration of Mongoose OS with Google IoT Core.

See tutorial at <https://mongoose-os.com/docs/mongoose-os/cloud/google.md>

Neopixel driver for Mongoose OS

RPC Service - Config

This service provides an ability to manage device configuration remotely.

It is required by the `mos config-get` and `mos config-set` commands.

If this library is not included in the app, those commands won't work.

It is possible to call this service programmatically via serial, HTTP/RESTful,

Websocket, MQTT or other transports

(see [RPC section](#)) or use `mos` tool.

Below is a list of exported RPC methods and arguments:

Config.Get

Get device configuration subtree. Arguments:

```
{  
    // Optional. Path to a config object, e.g. `wifi.sta.ssid`.  
    // If not specified, a full configuration tree is returned.  
    "key": "..."  
}
```

Example usage:

```
mos call Config.Get  
{  
    "http": {  
        "enable": true,  
        "listen_addr": "80",  
        ...  
    }  
}  
mos call Config.Get '{"key": "wifi.sta.enable"}'  
true
```

This RPC command has a shortcut: `mos config-get`:

```
mos config-get
{
  "http": {
    "enable": true,
    "listen_addr": "80",
    ...
mos config-get wifi.sta.enable
true
```

Config.Set

Set device configuration parameters. Arguments:

```
{
  // Required. Contains a sparse object with configuration parameters.
  // These parameters are applied on top of the existing device configuration.
  "config": { ... }
}
```

Example usage - set `debug.level` to 3:

```
mos call Config.Set '{"config": {"debug": {"level": 3}}}'
```

This RPC command has a shortcut: `mos config-set` which sets the config option, saves it, and reboots the device (since some config options take effect only after reboot):

```
mos config-set debug.level=3
Getting configuration...
Setting new configuration...
Saving and rebooting...
```

Config.Save

Writes an existing device configuration on flash, as a sequence of `confX.json` files (see [description](#)). This makes configuration permanent, preserved after device reboot. Arguments:

```
{
  "reboot": false // Optional. Whether to reboot the device after the call
}
```

Example usage:

```
mos call Config.Save '{"reboot": true}'
```

Core meta-library

Web UI for WiFi setup

Allows to call local RPC services - implements RPC

Loopback channel

This RPC service provides a way to call local RPC services - i.e. those running on the device itself.

Example usage in C/C++:

```
#include "mg_rpc_channel_loopback.h"
struct mg_rpc_call_opts opts = {.dst = mg_mk_str(MGOS_RPC_LOOPBACK_ADDR) };
mg_rpc_callf(mgos_rpc_get_global(), mg_mk_str("My.Func"), NULL, NULL, &opts,
             "{param1: %Q, param2: %d}", "jaja", 1234);
```

Example usage in JavaScript:

```
RPC.call(RPC.LOCAL, 'Config.Save', {reboot: true}, function (resp, ud) {
    print('Response:', JSON.stringify(resp));
}, null);
```

A cross-cloud device shadow API

Overview

This library provides a device shadow interface to manage device metadata on the cloud.

VL53L0X library for Mongoose OS

Version: 1.0

Release date: 2017 Nov 17

Summary

This is a library for the Mongoose OS that helps interface with ST's

[VL53L0X time-of-flight distance sensor](#).

The library makes it simple to configure the sensor and read range data from it via I²C.

This library is simple port of pololu's

[VL53L0X library for Arduino](#).

Supported platforms

This library is designed to work with Mongoose OS versions 1.19.6 or later;

I have not tested it with earlier versions.

This library should support any boards which support Mongoose OS and I²C.

Getting started

Hardware

I have tested with [ESPr Developer 32](#)

and [VL53L0X Time-of-Flight Range sensor module](#).

Make the following connections between mongoose os board and the VL53L0X module:

ESP32 boards

ESP32	VL53L0X
-----	-----
3V3	- VIN
GND	- GND
SDA(IO21)	- SDA
SCL(IO22)	- SCL

Software

Add configuration below to your application's mos.yml

```
config_schema:  
- ["i2c.enable", true]  
- ["i2c.sda_gpio", 21]  
- ["i2c.scl_gpio", 22]  
  
libs:  
- origin: https://github.com/mongoose-os-libs/arduino-pololu-vl53l0x
```

Examples

Simple Javascript example is available under examples directory.

ST's VL53L0X API and this library

Most of the functionality of this library is based on the

[VL53L0X API](#)

provided by ST (STSW-IMG005), and some of the explanatory comments in the code are quoted or paraphrased from the API source code, API user manual (UM2039), and the VL53L0X datasheet. For more explanation about the library code and how it was derived from the API, see the comments in VL53L0X.cpp.

This library is intended to provide a quicker and easier way to get started using the VL53L0X with an Arduino-compatible controller, in contrast to customizing and compiling ST's API for the Arduino. The library has a more streamlined interface, as well as smaller storage and memory footprints. However, it does not implement some of the more advanced functionality available in the API (for example, calibrating the sensor to work well under a cover glass), and it has less robust error checking. For advanced applications, especially when storage and memory are less of an issue, consider using the VL53L0X API directly.

Library reference

- `Pololu_VL53L0X.create()`

Constructor.

- `int getLastStatus()`

The status of the last I²C write transmission. See the [Wire.endTransmission\(\) documentation](#)

for return values.

- `void setAddress(int new_addr)`

Changes the I²C slave device address of the VL53L0X to the given value (7-bit).

- `int getAddress()`

Returns the current I²C address.

- `void begin()`
- `void end()`

Begin or end underlying Arduino's I²C library. Call begin before call init.

- `bool init_2v8()`
- `bool init_1v8()`

Initializes and configures the sensor. If the optional argument `io_2v8` is true (the default if not specified), the sensor is configured for 2V8 mode (2.8 V I/O); if false, the sensor is left in 1V8 mode. The return value is a boolean indicating whether the initialization completed successfully.

- `void writeReg(int reg, int value)`

Writes an 8-bit sensor register with the given value.

Register address constants are defined by the `regAddr` enumeration type in `VL53L0X.h`.

Example use: `sensor.writeReg(VL53L0X::SYSRANGE_START, 0x01);`

- `void writeReg16Bit(int reg, int value)`

Writes a 16-bit sensor register with the given value.

- `void writeReg32Bit(int reg, int value)`

Writes a 32-bit sensor register with the given value.

- `int readReg(int reg)`

Reads an 8-bit sensor register and returns the value read.

- `int readReg16Bit(int reg)`

Reads a 16-bit sensor register and returns the value read.

- `int readReg32Bit(int reg)`

Reads a 32-bit sensor register and returns the value read.

- `void writeMulti(int reg, char const * src, int count)`

Writes an arbitrary number of bytes from the given array to the sensor, starting at the given register.

- `void readMulti(int reg, int * dst, int count)`

Reads an arbitrary number of bytes from the sensor, starting at the given register, into the given array.

- `bool setSignalRateLimit(float limit_Mcps)`

Sets the return signal rate limit to the given value in units of MCPS (mega counts per second). This is the minimum amplitude of the signal reflected from the target and received by the sensor necessary for it to report a valid reading. Setting a lower limit increases the potential range of the sensor but also increases the likelihood of getting an inaccurate reading because of reflections from objects other than the intended target. This limit is initialized to 0.25 MCPS by default. The return value is a boolean indicating whether the requested limit was valid.

- `float getSignalRateLimit()`

Returns the current return signal rate limit in MCPS.

- `bool setMeasurementTimingBudget(int budget_us)`

Sets the measurement timing budget to the given value in microseconds. This is the time allowed for one range measurement; a longer timing budget allows for more accurate measurements. The default budget is about 33000 microseconds, or 33 ms; the minimum is 20 ms. The return value is a boolean indicating whether the requested budget was valid.

- `int getMeasurementTimingBudget()`

Returns the current measurement timing budget in microseconds.

- `bool setVcSELpulsePeriod(vcSELPeriodType type, int period_pclks)`

Sets the VCSEL (vertical cavity surface emitting laser) pulse period for the given period type (`VL53L0X::VcSELPeriodPreRange` or `VL53L0X::VcSELPeriodFinalRange`) to the given value (in PCLKs). Longer periods increase the potential range of the sensor. Valid values are (even numbers only):

Pre: 12 to 18 (initialized to 14 by default)

Final: 8 to 14 (initialized to 10 by default)

The return value is a boolean indicating whether the requested period was valid.

- `int getVcSELpulsePeriod(vcSELPeriodType type)`

Returns the current VCSEL pulse period for the given period type.

- `void startContinuous(int period_ms = 0)`

Starts continuous ranging measurements. If the optional argument `period_ms` is 0 (the default if not specified), continuous back-to-back mode is used (the sensor takes measurements as often as possible); if it is nonzero, continuous timed mode is used, with the specified inter-measurement period in milliseconds determining how often the sensor takes a measurement.

- `void stopContinuous()`

Stops continuous mode.

- `int readRangeContinuousMillimeters()`

Returns a range reading in millimeters when continuous mode is active.

- `int readRangeSingleMillimeters()`

Performs a single-shot ranging measurement and returns the reading in millimeters.

- `void setTimeout(int timeout)`

Sets a timeout period in milliseconds after which read operations will abort if the sensor is not ready. A value of 0 disables the timeout.

- `int getTimeout()`

Returns the current timeout period setting.

- `bool timeoutOccurred()`

Indicates whether a read timeout has occurred since the last call to `timeoutOccurred()`.

Version history

- 1.0 (2017 Nov 17): Original release.
- 1.1 (2017 Nov 29): Avoid "unused-parameter" error.

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

HTU21D(F) I2C Driver

A Mongoose library for Measurement Specialities HTU21D(F) integrated circuit.

Sensor details

The HTU21D(F) is a new digital humidity sensor with temperature output by MEAS. Setting new standards in terms of size and intelligence, it is embedded in a reflow solderable Dual Flat No leads (DFN) package with a small 3 x 3 x 0.9 mm footprint. This sensor provides calibrated, linearized signals in digital, I2C format.

HTU21D(F) digital humidity sensors are dedicated humidity and temperature plug and play transducers for OEM applications where reliable and accurate measurements are needed. Direct interface with a micro-controller is made possible with the module for humidity and temperature digital outputs. These low power sensors are designed for high volume and cost sensitive applications with tight space constraints.

Every sensor is individually calibrated and tested. Lot identification is printed on the sensor and an electronic identification code is stored on the chip – which can be read out by command. Low battery can be detected and a checksum improves communication reliability. The resolution of these digital humidity sensors can be changed by command (8/12bit up to 12/14bit for RH/T).

With MEAS' improvements and miniaturization of this sensor, the performance-to-price ratio has been improved – and eventually, any device should benefit from its cutting edge energy saving operation mode. Optional PTFE filter/membrane (F) protects HTU21D digital humidity sensors against dust and water immersion, as well as against contamination by particles. PTFE filter/membranes preserve a high response time. The white PTFE filter/membrane is directly stuck on the sensor housing.

See [datasheet](#)

for implementation details.

A great place to buy a ready made and tested unit is at [Adafruit](#).

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```
#include "mgos.h"
#include "mgos_i2c.h"
#include "mgos_htu21df.h"

static struct mgos_htu21df *s_htu21df;

static void timer_cb(void *user_data) {
    float temperature, humidity;

    temperature=mgos_htu21df_getTemperature(s_htu21df);
    humidity=mgos_htu21df_getHumidity(s_htu21df);

    LOG(LL_INFO, ("htu21df temperature=%.2f humidity=%.2f", temperature, humidity));

    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_i2c *i2c;

    i2c=mgos_i2c_get_global();
    if (!i2c) {
        LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
    } else {
        s_htu21df=mgos_htu21df_create(i2c, 0x40); // Default I2C address
        if (s_htu21df) {
            mgos_set_timer(5000, true, timer_cb, NULL);
        } else {
            LOG(LL_ERROR, ("Could not initialize sensor"));
        }
    }
    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

Si7021 I2C Driver

A Mongoose library for Silicon Si7021 integrated circuit.

Sensor details

The Si7021 I2C Humidity and Temperature Sensor is a monolithic CMOS IC integrating humidity and temperature sensor elements, an analog-to-digital converter, signal processing, calibration data, and an I2C Interface. The patented use of industry-standard, low-K polymeric dielectrics for sensing humidity enables the construction of low-power, monolithic CMOS Sensor ICs with low drift and hysteresis, and excellent long term stability.

The humidity and temperature sensors are factory-calibrated and the calibration data is stored in the on-chip non-volatile memory. This ensures that the sensors are fully interchangeable, with no recalibration or software changes required.

The Si7021 is available in a 3x3 mm DFN package and is reflow solderable. It can be used as a hardware- and software-compatible drop-in upgrade for existing RH/temperature sensors in 3x3 mm DFN-6 packages, featuring precision sensing over a wider range and lower power consumption. The optional factory-installed cover offers a low profile, convenient means of protecting the sensor during assembly (e.g., reflow soldering) and throughout the life of the product, excluding liquids (hydrophobic/oleophobic) and particulates.

The Si7021 offers an accurate, low-power, factory-calibrated digital solution ideal for measuring humidity, dew-point, and temperature, in applications ranging from HVAC/R and asset tracking to industrial and consumer platforms.

See [datasheet](#)

for implementation details.

A great place to buy a ready made and tested unit is at [Adafruit](#).

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```

#include "mgos.h"
#include "mgos_i2c.h"
#include "mgos_si7021.h"

static struct mgos_si7021 *s_si7021;

static void timer_cb(void *user_data) {
    float temperature, humidity;

    temperature=mgos_si7021_getTemperature(s_si7021);
    humidity=mgos_si7021_getHumidity(s_si7021);

    LOG(LL_INFO, ("si7021 temperature=%.2f humidity=%.2f", temperature, humidity));

    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_i2c *i2c;

    i2c=mgos_i2c_get_global();
    if (!i2c) {
        LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
    } else {
        s_si7021=mgos_si7021_create(i2c, 0x40); // Default I2C address
        if (s_si7021) {
            mgos_set_timer(5000, true, timer_cb, NULL);
        } else {
            LOG(LL_ERROR, ("Could not initialize sensor"));
        }
    }
    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Arduino Adafruit TSL2561 library for Mongoose OS

This is a port of the [Adafruit Arduino driver for the TSL2561 sensor](#) ported to run on the [Mongoose OS ecosystem](#).

Usage is extremely simple....

in ***mos.yml***, add to **libs:** section,

- origin: <https://github.com/mongoose-os-libs/arduino-adafruit-tsl2561>

in your ***init.js***, add something like the following,

```
load('api_arduino_tsl2561.js');
```

and

```
//Initialize Adafruit_TSL2561 library
let tsl = Adafruit_TSL2561.create();
print('Adafruit_TSL2561.TSL2561_GAIN_16X -> ',Adafruit_TSL2561.TSL2561_GAIN_16X);
tsl.setGain(Adafruit_TSL2561.TSL2561_GAIN_16X);
tsl.setIntegrationTime(Adafruit_TSL2561.TSL2561_INTEGRATIONTIME_402MS);
tsl.begin();

let tslGetData = function() {
  let vis = tsl.getVisible();
  let ir = tsl.getInfrared();
  let lux = tsl.calculateLux(vis, ir);
  print('TSL2561: Vis: ', vis, ', IR: ', ir, ', Lux: ', lux);
};

let tslTimer = Timer.set(10000 /* milliseconds */, true /* repeat */, tslGetData, null);
```

to use the library.

Enjoy!

This library provides PWM ([pulse-width modulation](#)) support for Mongoose OS.

A Library of Utility Functions for Mongoose OS apps with Sensors

Overview

A collection of C-functions with JS wrappers that make it convenient to calculate derived quantities (e.g. dew point temperature), convert between different units of measure, and perform simple statistics on arrays of (double precision) values.

Typical usage pattern:

in ***mos.yml***, add to **libs:** section,

```
- origin: https://github.com/mongoose-os-libs/sensor-utils
```

in your ***init.js***, add something like the following,

```
load('api_sensor_utils.js');
```

and (here is example usage that tests the API),

```

print('Test api_sensor_utils.js          ');

//Initialize a statistics object (handle)

let StatsObj = SensorUtils.initStatistics(4);

// add some numbers (e.g. obtained from sensor readings)

SensorUtils.addDataValue( StatsObj, 5.51 );
SensorUtils.addDataValue( StatsObj, 8.43 );
SensorUtils.addDataValue( StatsObj, 6.77 );
SensorUtils.addDataValue( StatsObj, 4.33 );
SensorUtils.addDataValue( StatsObj, 6.20 );
SensorUtils.addDataValue( StatsObj, 7.18 );
SensorUtils.addDataValue( StatsObj, 4.92 );
SensorUtils.addDataValue( StatsObj, 5.89 );

print('Statistics: ',SensorUtils.calculateStatistics(StatsObj));

// other convenience functions.

print('SensorUtils.fahrenheit(0.0) -> ',SensorUtils.fahrenheit(0.0));
print('SensorUtils.celsius(32.0) -> ',SensorUtils.celsius(32.0));
print('SensorUtils.inchesHg(101325) -> ',SensorUtils.inchesHg(101325));
print('SensorUtils.mmHg(101325) -> ',SensorUtils.mmHg(101325));
print('SensorUtils.atmospheresHg(29.9213) -> ',SensorUtils.atmospheresHg(29.9213));
print('SensorUtils.atmospheresP(101325) -> ',SensorUtils.atmospheresP(101325));
print('SensorUtils.lengthF(1/12) -> ',SensorUtils.lengthF(1.0/12.0));
print('SensorUtils.computeDewpoint(4.6, 50.89) -> ',SensorUtils.computeDewpoint(4.6, 50.89));
print('SensorUtils.computeAltitude(103245.5, 101325.0) -> ',SensorUtils.computeAltitude(103245.5, 101325.0));

```

Executing the above JS code on an ESP32 typically gives the following output:

```

[Jan 17 08:35:35.996] Test api_sensor_utils.js

[Jan 17 08:35:36.183] Statistics: {"samples":8, "length":8, "mean":6.1538, "std_deviation":1.3065, "data":[ 5.5100, 8.4300, 6.7700, 4.3300, 6.2000, 7.1800, 4.9200, 5.8900]}

[Jan 17 08:35:36.214] SensorUtils.fahrenheit(0.0) -> 32
[Jan 17 08:35:36.226] SensorUtils.celsius(32.0) -> 0
[Jan 17 08:35:36.239] SensorUtils.inchesHg(101325) -> 29.921252
[Jan 17 08:35:36.255] SensorUtils.mmHg(101325) -> 1.178002
[Jan 17 08:35:36.268] SensorUtils.atmospheresHg(29.9213) -> 1.000043
[Jan 17 08:35:36.282] SensorUtils.atmospheresP(101325) -> 1
[Jan 17 08:35:36.295] SensorUtils.lengthF((2.54/100.0) * 12) -> 1.000000
[Jan 17 08:35:36.313] SensorUtils.computeDewpoint(4.6, 50.89) -> -4.634330
[Jan 17 08:35:36.330] SensorUtils.computeAltitude(103245.5, 101325.0) -> -158.677439

```

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own

the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

Unit tests MGOS Application

This directory contains some small unit tests for the application logic. It is meant to compile and run on the host operating system, in the author's case this is Ubuntu LTS. To run the test suite:

```
make  
./test
```

prometheus-metrics

A Mongoose OS Prometheus Metrics library.

Introduction

[Prometheus](#) is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has

a very active developer and user community. It is now a standalone open source project and maintained independently of any company.

[Mongoose OS](#) is a purpose-built secure Operating System for commercial connected devices. It focuses on stable and secure functioning of multiple connected devices in production and post-sale stages. Key features include secure communication (TLS), over-the-air updates (OTA) and remote device management. These features are usually missing from SDK and their correct implementation would be a complex and resource consuming task. Neglecting them may result in compromised device security and negative brand perception of your products.

Structure

`prometheus-metrics` is a library component that can be added to the app's `mos.yml` file without any configuration needed out of the box, and it pulls in the `http-server` module. The library opens a `/metrics` endpoint which exposes the operating system and library vitalsigns to Prometheus.

By adding the library to the build manifest in `mos.yml`, a compiler define `MGOS_HAVE_PROMETHEUS_METRICS` is set, which other libraries can use to create metrics and update them. This is *non intrusive* because if the library is not used, no additional code is compiled in Mongoose OS and its libraries.

Implementation

Base MGOS Metrics

All Mongoose vitals (memory, WiFi/Ethernet, CPU, scheduling) are exposed using the `mgos_` prefix.

```
# HELP mgos_build Build info
# TYPE mgos_build gauge
mgos_build{app="empty",id="20171121-164823/???",version="1.1.04"} 1
# HELP mgos_platform Platform information
# TYPE mgos_platform gauge
mgos_platform{arch="esp32",mac="240AC4106560",idf="v1.0-2815-g50a73c1"} 1
# HELP mgos_uptime Uptime in seconds
# TYPE mgos_uptime counter
mgos_uptime 1888
# HELP mgos_heap_size System memory size
# TYPE mgos_heap_size gauge
mgos_heap_size 295076
```

Platform Specific Metrics

Platform specific vitals are exposed using the `$platform_` prefix, for example `esp32_` for ESP32 and ESP-IDF metrics.

```
# HELP esp32_chip_info ESP32 Chip Information
# TYPE esp32_chip_info gauge
esp32_chip_info{model=0,cores=2,revision=1,features=32,sdk="master"} 1
# HELP esp32_num_tasks ESP32 FreeRTOS task count
# TYPE esp32_num_tasks gauge
esp32_num_tasks 9
```

Library Specific Metrics

Library owners gate the code that creates, updates and exposes the metrics

by the define `MGOS_HAVE_PROMETHEUS_METRICS`. Metrics should be defined as static variables to stay private to the implementation. Then, a callback function is installed, and `prometheus-metrics` will loop over all registered callbacks to allow them to add their metrics to the output.

Taking `mqtt` as an example:

```
#if MGOS_HAVE_PROMETHEUS_METRICS
#include "mgos_prometheus_metrics.h"

static uint32_t metrics_mqtt_sent_topics_count = 0;
static uint32_t metrics_mqtt_sent_topics_bytes_total = 0;
static uint32_t metrics_mqtt_received_topics_count = 0;
static uint32_t metrics_mqtt_received_topics_bytes_total = 0;

static void metrics_mqtt(struct mg_connection *nc, void *user_data) {
    mgos_prometheus_metrics_printf(nc, COUNTER,
        "mgos_mqtt_sent_topics_count", "MQTT topics sent",
        "%u", metrics_mqtt_sent_topics_count);

    mgos_prometheus_metrics_printf(nc, COUNTER,
        "mgos_mqtt_sent_topics_bytes_total", "Total bytes sent in MQTT topics",
        "%u", metrics_mqtt_sent_topics_bytes_total);

    mgos_prometheus_metrics_printf(nc, COUNTER,
        "mgos_mqtt_received_topics_count", "MQTT topics sent",
        "%u", metrics_mqtt_received_topics_count);

    mgos_prometheus_metrics_printf(nc, COUNTER,
        "mgos_mqtt_received_topics_bytes_total", "Total bytes received in MQTT topics",
        "%u", metrics_mqtt_received_topics_bytes_total);

    (void) user_data;
}

#endif // MGOS_HAVE_PROMETHEUS_METRICS
```

Then in the library's `init` function, register the callback:

```
bool mgos_mqtt_init(void) {
    #if MGOS_HAVE_PROMETHEUS_METRICS
        mgos_prometheus_metrics_add_handler(metrics_mqtt, NULL);
    #endif
    return true;
}
```

As mentioned above, if the `prometheus-metrics` library is not included in the app's `mos.yml` manifest, no code will be compiled which makes the addition *non intrusive*.

Application Specific Metrics

Users are able to add their app's own metrics in the same way as libraries can. They do this by registering a handler function, which is called from `prometheus-metrics`.

```
#include "mgos_prometheus_metrics.h"
uint32_t my_counter=0;

static void prometheus_metrics_fn(struct mg_connection *nc, void *user_data) {
    mgos_prometheus_metrics_printf(nc, COUNTER,
        "my_counter", "Total things counted",
        "%u", my_counter);
    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_prometheus_metrics_add_handler(prometheus_metrics_fn, NULL);
    return MGOS_APP_INIT_SUCCESS;
}
```

POSTing to Pushgateway

Prometheus offers an intermediate receiver called a [Pushgateway](#), see their

[codebase](#) for details. Some users

may not wish to have their Mongoose IoT device listen on the network for HTTP connections to the `/metrics` endpoint, for security reasons. As an alternative, the library can be configured to close its serving endpoint, and push its metrics upstream instead. In `mos.yml`:

```
config_schema:
- ["prometheus.server_enable", false]
- ["prometheus.pushgateway", "s", "example.com:9091"]
```

An example program using a timer to POST every 5 seconds:

```
#include "mgos.h"
#include "mgos_prometheus_metrics.h"

static void timer_cb(void *user_data) {
    mgos_prometheus_metrics_push("test1", "instance1");
    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_set_timer(5000, true, timer_cb, NULL);
    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

FreeRTOS Kernel

This is official FreeRTOS 10.2.0 kernel from www.freertos.org with some irrelevant stuff removed.

It is used by all the ports except ESP32, which uses its own, heavily modified version.

A device provision library

This library provides the following functionality:

Reset to factory defaults on button press

This functionality allows to reset the device to factory defaults by performing the following steps:

- remove `conf9.json` which holds user-specific device configuration
- perform filesystem garbage collection
- reboot the device

Reset to factory defaults is done by configuring a reset button, attached to a certain GPIO pin. Two modes are possible:

- Hold the reset button and reboot the device while holding a button.
For this, set `provision.btn.pin` to a non-negative value, and set `provision.btn.hold_ms` to 0.
- Just hold the reset button pressed for a long time.
For this, set `provision.btn.pin` to a non-negative value, and set `provision.btn.hold_ms` to a positive value, e.g. 5000 - 5 seconds.
If you also set `provision.btn.inhibit_after_s` value to non-zero number of seconds, reset button will be inhibited once uptime reaches this value (reset button will stop working N seconds after boot).

Reset to factory defaults on bad WiFi credentials

This functionality resets the device to factory defaults if the WiFi configuration is entered by user incorrectly (wrong network name or wrong password).

If the device has at least once connected to the WiFi station successfully, reset is never done afterwards.

This is done via configuration parameter `provision.wifi_configured`, which is set to `false` on a fresh device. When the device first connects to the WiFi station, `provision.wifi_configured` is set to `true`. When WiFi connection fails, and `provision.wifi_configured` is `false`, factory reset is triggered.

Provisioning state

Device's provisioning state is tracked as two variables: current and maximum.

Both start at 0 but the current state resets on each boot and maximum state is persisted across reboots.

Maximum state can be used to ensure successful provisioning by specifying a "stable state" (`provision.stable_state`) and a timeout (`provision.timeout`).

If stable state is not reached within the specified time, device config is wiped and device is rebooted (which also resets max state to 0).

Four states are pre-defined:

- Unprovisioned (0). Nothing is configured, will stay in this state without reboot no matter what stable state is configured.
- Networking configured, connecting (1)
- Connecting to cloud (2)
- Connected to cloud (3)

Default value for `provision.stable_state` is 3, so the device will assume it has been fully provisioned once connected to the cloud.

If cloud connection is not required, this can be lowered to 2. If set to 0, device reset is not performed.

Stable state higher than 3 can be used if additional provisioning steps after successful connection are required.

User application code will need to manage further provisioning and use `mgos_provision_set_cur_state()` to signal state transitions.

When state transition occurs an event is raised which application code can use to indicate current provisioning state.

LED indication

If `provision.led.pin` is configured, an event handler will be installed that will use LED to visually indicate the current provisioning state:

- Unprovisioned (0): Blink the LED slowly (once a second)
- Networking configured, connecting (1): Blink faster (every 0.5s)
- Connecting to cloud (2): Blink fast (every 0.25s)
- Connected to cloud (3, or whatever configured stable state is): Solid on

Configuration parameters reference

```
"provision": {  
    "configured": false, // Set to true when first time connected to WiFi  
    "button": {  
        "pin": 17, // Reset button GPIO number. Negative value disables reset  
        // button functionality. Default: arch-specific.  
        "hold_ms": 0, // Number of milliseconds to hold to trigger a factory reset.  
        // If negative, disable. If 0, reset on boot.  
        // If positive, reset when pressed for that many milliseconds.  
    }  
}
```

Hello JavaScript/mJS

Intro

This example shows how to use the library with JavaScript or rather mJS.

Prerequisite

Setup [Mongoose OS](https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md) as described here: <https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md>

Please note that the chosen configuration is for a [Adafruit HUZZAH ESP32](#) or a [Adafruit HUZZAH ESP8266](#), and a [\(monochrome\) ePaper Display FeatherWing](#). So please adjust the configuration (GPIOs, display width/height/driver, etc. in the `config_schema` section) in `mos.yml` accordingly for your setup.

Installation

```
git checkout https://github.com/bbilger/mgos-arduino-adafruit-epd  
cd mgos-arduino-adafruit-epd/examples/hello_js  
  
mos build --local --platform esp32 && mos flash && mos console  
# or  
mos build --local --platform esp8266 && mos flash && mos console
```

Hello C++

Intro

This example shows how to use the library with C++.

Prerequisite

Setup [Mongoose OS](https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md) as described here: <https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md>

Please note that the chosen configuration is for a [Adafruit HUZZAH ESP32](#) or a [Adadafruit HUZZAH ESP8266](#), and a [\(monochrome\) ePaper Display FeatherWing](#). So please adjust the configuration (GPIOs, display width/height/driver, etc. in the config_schema section) in `mos.yml` accordingly for your setup.

Installation

```
git checkout https://github.com/bbilger/mgos-arduino-adafruit-epd
cd mgos-arduino-adafruit-epd/examples/hello_cpp

mos build --local --platform esp32 && mos flash && mos console
# or
mos build --local --platform esp8266 && mos flash && mos console
```

Hello C

Intro

This example shows how to use the library with C.

Prerequisite

Setup [Mongoose OS](https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md) as described here: <https://mongoose-os.com/docs/mongoose-os/quickstart/setup.md>

Please note that the chosen configuration is for a [Adafruit HUZZAH ESP32](#) or a [Adadafruit HUZZAH ESP8266](#), and a [\(monochrome\) ePaper Display FeatherWing](#). So please adjust the configuration (GPIOs, display width/height/driver, etc. in the config_schema section) in `mos.yml` accordingly for your setup.

Installation

```
git checkout https://github.com/bbilger/mgos-arduino-adafruit-epd
cd mgos-arduino-adafruit-epd/examples/hello_c

mos build --local --platform esp32 && mos flash && mos console
# or
mos build --local --platform esp8266 && mos flash && mos console
```

Adafruit E-Paper / E-Ink Library for Mongoose OS

Introduction

This is a port of or rather a wrapper around the [Adafruit EPD Library](#) for [Mongoose OS](#).

Please check [Adafruit EPD Library](#) for additional information
and also please note the original [README](#).

Notes:

- this was moved from [bbilger/mgos-arduino-adafruit-epd](#) which was forked from [adafruit/Adafruit_EPD](#)
- EPD stands for e-paper display.

Usage

Examples

There are three examples that show how to use this library:

- [C usage example](#)
- [JavaScript or rather mJS usage example](#)
- [C++ usage example](#)

C

```
#include "mgos_aepd.h"
...
// get the e-paper display as configured in mos.yml
Adafruit_EPD *epd = mgos_aepd_get_global();
// methods of the Adafruit_EPD C++ class
// can be invoked through "mgos_aepd_<snake_case_method_name>(epd, ...)"
mgos_aepd_some_method(epd, ...);
...
```

JavaScript / mJS

```
load('api_arduino_adafruit_epd.js');
// get the e-paper display as configured in mos.yml
let epd = ArduinoAdafruitEpd.createGlobal();
// the JavaScript object has the same / most methods the original Adafruit_EPD C++ class has
epd.someMethod(...);
```

C++

```
#include "Adafruit_EPD.h"
#include "mgos_aepd.h"

// get the e-paper display as configured in mos.yml
Adafruit_EPD *epd = mgos_aepd_get_global();
// nothing fancy here, since one can use the original Adafruit_EPD C++ class directly
epd.someMethod(...);
```

Configuration

It's optional to setup the display using the following config but unless you are using more than one display this is probably what you want.

config	type	default	required	comment
aepd.enable	bool	false	-	If enabled, then the display will be setup automatically by the library and the rest of the config must be valid.
aepd.begin	bool	true	-	Calls begin on the display automatically on start.
aepd.driver	string	empty	true	The driver to use for the connected display. At the moment the following drivers are implemented: "IL0373", "IL0398", "IL91874", "SSD1608", "SSD1675".
aepd.width	int	-1	true	Display width in pixels.
aepd.height	int	-1	true	Display height in pixels.
aepd.epd_spi_cs_index	int	-1	true	spi.csX_gpio index for the EDP, 0, 1 or 2.
aepd.sram_spi_cs_index	int	-1	false	spi.csX_gpio index for the SRAM, 0, 1 or 2, or -1 to not use SRAM.
aepd.epd_dc_gpio	int	-1	true	EPD DC GPIO.
aepd.epd_reset_gpio	int	-1	false	EPD reset GPIO or -1
aepd.debug	bool	false	-	At the moment it only prevents a bootloop on misconfiguration.

If you whish to not setup the display via configuration, then you can set it up in C via `mgos_aepd_create`, in C++ by directly instantiating a concrete subclass of `Adafruit_EPD` (e.g. `Adafruit_SSD1675`), and in JS you are on your own but you can for example "ffi" `mgos_aepd_create`.

License

This library is - like the original one - licensed under [MIT license](#).

Adafruit EPD Library

[build passing](#)

This is a library for the Adafruit E-paper displays:

- <https://www.adafruit.com/products/3625>

Check out the links above for our tutorials and wiring diagrams. These devices use SPI to communicate

Adafruit invests time and resources providing this open source code, please support Adafruit and open-source hardware by purchasing products from Adafruit!

Written by Dean Miller for Adafruit Industries.

MIT license, all text above must be included in any redistribution

Implementation of Mongoose OS OTA HTTP server

Overview

This library adds several HTTP endpoints that allow to POST new firmware to the device over HTTP:

- /update - accept new firmware uploads via HTTP POST.
- /update/revert - roll back to the previous firmware.
- /update/commit - commit new firmware.

Example using curl tool (use Mac/Linux terminal or Windows command prompt).

Assume you have build a new firmware for your app. The zip file with a built firmware is located at build/fw.zip. In order to update a live device with IP address IP_ADDRESS, do:

```
$ curl -i -F filedata=@./build/fw.zip http://IP_ADDRESS/update
HTTP/1.1 200 OK
Server: Mongoose/6.10
Content-Type: text/plain
Connection: close
```

Update applied, finalizing

IR protocol library

Usage

Either

src/main.c

```
#include "mgos.h"
#include "mgos_ir.h"

static void irrecv_cb(int code, void *arg)
{
    LOG(LL_INFO, ("IR: %08X", code));
    (void) arg;
}

enum mgos_app_init_result mgos_app_init(void)
{
    // TSOP on pin 14, NEC protocol
    mgos_irrecv_nec_create(14, irrecv_cb, NULL);

    return MGOS_APP_INIT_SUCCESS;
}
```

or

fs/init.js

```
load("api_ir.js");

// TSOP on pin 14, NEC protocol
let ir = IR.Receiver.NEC.create(14, function(code) {
    print("IR", code);
}, null);
```

MAX72xx LED Matrix Display Library

IMPORTANT NOTE: Please make sure that you find and read the html documentation that comes with the library (open "docs/index.html") or use the link below. **You will need to edit the MAX72xx.h file to configure the type of matrix you are using.** This is the most asked support question so avoid frustration and READ THE MANUAL in the docs subfolder.

The library implements functions that allow the MAX72xx to be used for LED matrices (64 individual LEDs), allowing the programmer to use the LED matrix as a pixel device, displaying graphics elements much like any other pixel addressable display.

In this scenario, it is convenient to abstract out the concept of the hardware device and create a uniform and consistent pixel address space, with the libraries determining device and device-element address. Similarly, control of the devices is uniform and abstracted to a system level.

The library still retains flexibility for device level control, should the developer require, through the use of overloaded class methods.

[Library Documentation](#)

RPC Service - OTA (Over The Air updates)

This service provides an ability to manage OTA on devices remotely.

It is possible to call this service programmatically via serial, HTTP/RESTful, Websocket, MQTT or other transports (see [RPC section](#)) or via the `mos` tool.

See in-depth description of our OTA mechanism at

[Updating firmware reliably - embedded.com](#).

See OTA video tutorial:

Below is a list of exported RPC methods and arguments:

OTA.Update

Trigger OTA firmware update. Arguments:

```
{  
  "url": "https://foo.com/fw123.zip", // Required. URL to the new firmware.  
  "commit_timeout": "300"           // Optional. Time frame in seconds to do  
                                // OTA.Commit after reboot. If commit is  
                                // not done during the timeout, OTA rolls back.  
}
```

A new firmware gets downloaded to the separate flash partition, and is marked dirty. When the download is complete, device is rebooted. After reboot, a firmware partition could become committed by calling `OTA.Commit` - in which case, it is marked as "good". Otherwise, a device

reboots back into the old firmware after the `commit_timeout` seconds.

Example usage:

```
mos call OTA.Update '{"url": "http://1.2.3.4/fw.zip", "commit_timeout": 300}'
```

OTA.Commit

Commit current firmware. Arguments: none.

Example usage:

```
mos call OTA.Commit
```

OTA.Revert

Rolls back to the previous firmware. Arguments: none.

Example usage:

```
mos call OTA.Revert
```

OTA.CreateSnapshot

Create new firmware partition with the copy of currently running firmware. Arguments:

```
{
    // Optional. If true, then current firmware is uncommitted, and needs to
    // be explicitly committed after the first reboot. Otherwise, it'll reboot
    // into the created snapshot. This option is useful if a dangerous, risky
    // live update is to be done on the living device. Then, if the update
    // fails and device bricks, it'll revert to the created good snapshot.
    "set_as_revert": false,
    // Optional. Same meaning as for OTA.Update
    "commit_timeout": "300"
}
```

Example usage:

```
mos call OTA.CreateSnapshot
```

OTA.GetBootState

Get current boot state. Arguments: none.

Example usage:

```
mos call OTA.GetBootState
{
    "active_slot": 0,      # Currently active flash partition.
    "is_committed": true, # Current firmware is marked as "good" (committed).
    "revert_slot": 0,      # If uncommitted, slot to roll back to.
    "commit_timeout": 0    # Commit timeout.
}
```

OTA.SetBootState

Get current boot state. Arguments: see `OTA.GetBootState` reply section.

Example usage:

```
mos call OTA.SetBootState '{"revert_slot": 1}'
```

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

Mongoose OS Barometer Library

Introduction

This library provides a simple API that describes barometer sensors. It implements various popular I2C and SPI barometers behind that API. Callers of the library can rely on the API returning consistently typed data regardless of the choice of sensor.

Primitives

(work in progress)

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

System Configuration over Bluetooth GATT Service

Overview

This library provides a way to examine and change configuration over Generic Attribute Service (GATT) Bluetooth Low-Energy (BLE) service.

The service is designed to be usable with any generic BLE mobile app that supports GATT, e.g. BLE Scanner ([Android](#), [iOS](#)).

Note: Default BT configuration is permissive. See <https://github.com/mongoose-os-libs/bt-common#security> for a better idea.

Attribute description

The service UUID is `5f6d4f53-5f43-4647-5f53-56435f49445f`, which is a representation of a 16-byte string `_mOS_CFG_SVC_ID_`.

The service defines 3 characteristics (attributes):

- `306d4f53-5f43-4647-5f6b-65795f5f30 (0mOS_CFG_key_0)` - a write-only attribute that selects the configuration key to be operated on. The key is a string of components separated by period, e.g. `wifi.sta.ssid` or `debug.level`.
- `316d4f53-5f43-4647-5f76-616c75655f31 (1mOS_CFG_value_1)` - a read-write attribute that returns value of the selected key when read or accepts the value to be set. Note that value is not applied immediately after writing. Instead, consecutive writes to this key are appended to form the value to be set (this is due to small value of the default MTU and differences in how clients handle writes exceeding the MTU). For easy manual entry, attribute values are returned as strings and expected as strings as well. So, to enter value of 123, one should submit string `123`, not hex value `0x7b`. Boolean values should be entered as strings `true` or `false`.
- `326d4f53-5f43-4647-5f73-6176655f5f32 (2mOS_CFG_save_2)` - a write-only attribute that applies value submitted to the value attribute to the key selected via the key attribute. The value written to this key can be one of:
 - `0` - just set the value, change will not be persisted
 - `1` - set and save the config
 - `2` - set, save and reboot

Example - configuring WiFi

Here is an example of the configuration provisioning a device - configuring WiFi station settings (we'll use `key`, `value` and `save` to abbreviate long UUIDs):

- Set the SSID, do not save the config yet
 - `wifi.sta.ssid` -> `key`
 - `MyNetwork` -> `value`
 - `0` -> `save`
- Set the password, do not save the config yet
 - `wifi.sta.pass` -> `key`
 - `MyPassword` -> `value`
 - `0` -> `save`
- Enable the station, save and reboot
 - `wifi.sta.enable` -> `key`
 - `true` -> `value`
 - `2` -> `save`

Once device is confirmed to have successfully connected to WiFi, BT config can be disabled:

- `bt.config_enable` -> `key` # Or `bt.enable` to disable Bluetooth entirely.
- `false` -> `value`
- `2` -> `save` # Save and reboot with BT configuration disabled.

See Also

See [rpc-gatts](#) library,

which provides a GATT service that acts as an RPC channel. It accepts incoming RPC frames and can send them as well - or rather, makes them available for collection.

BME680 sensor API

Introduction

This package contains the Bosch Sensortec's BME680 gas sensor API

The sensor driver package includes `bme680.h`, `bme680.c` and `bme680_defs.h` files

Version

File	Version	Date
<code>bme680.c</code>	3.5.8	22 Feb 2018
<code>bme680.h</code>	3.5.8	22 Feb 2018
<code>bme680_defs.h</code>	3.5.8	22 Feb 2018

Integration details

- Integrate `bme680.h`, `bme680_defs.h` and `bme680.c` file in to your project.
- Include the `bme680.h` file in your code like below.

```
#include "bme680.h"
```

File information

- bme680_defs.h : This header file has the constants, macros and datatype declarations.
- bme680.h : This header file contains the declarations of the sensor driver APIs.
- bme680.c : This source file contains the definitions of the sensor driver APIs.

Supported sensor interfaces

- SPI 4-wire
- I2C

Usage guide

Initializing the sensor

To initialize the sensor, you will first need to create a device structure. You can do this by creating an instance of the structure bme680_dev. Then go on to fill in the various parameters as shown below

Example for SPI 4-Wire

```
struct bme680_dev gas_sensor;

/* You may assign a chip select identifier to be handled later */
gas_sensor.dev_id = 0;
gas_sensor.intf = BME680_SPI_INTF;
gas_sensor.read = user_spi_read;
gas_sensor.write = user_spi_write;
gas_sensor.delay_ms = user_delay_ms;

/* amb_temp can be set to 25 prior to configuring the gas sensor
 * or by performing a few temperature readings without operating the gas sensor.
 */
gas_sensor.amb_temp = 25;

int8_t rslt = BME680_OK;
rslt = bme680_init(&gas_sensor);
```

Example for I2C

```
struct bme680_dev gas_sensor;

gas_sensor.dev_id = BME680_I2C_ADDR_PRIMARY;
gas_sensor.intf = BME680_I2C_INTF;
gas_sensor.read = user_i2c_read;
gas_sensor.write = user_i2c_write;
gas_sensor.delay_ms = user_delay_ms;

/* amb_temp can be set to 25 prior to configuring the gas sensor
 * or by performing a few temperature readings without operating the gas sensor.
 */
gas_sensor.amb_temp = 25;

int8_t rslt = BME680_OK;
rslt = bme680_init(&gas_sensor);
```

Regarding compensation functions for temperature, pressure, humidity and gas we have two implementations.

- Integer version
- floating point version

By default, Integer version is used in the API

If the user needs the floating point version, the user has to un-comment BME680_FLOAT_POINT_COMPENSATION macro in bme680_defs.h file or to add it in the compiler flags.

Configuring the sensor

Example for configuring the sensor in forced mode

```
uint8_t set_required_settings;

/* Set the temperature, pressure and humidity settings */
gas_sensor.tph_sett.os_hum = BME680_OS_2X;
gas_sensor.tph_sett.os_pres = BME680_OS_4X;
gas_sensor.tph_sett.os_temp = BME680_OS_8X;
gas_sensor.tph_sett.filter = BME680_FILTER_SIZE_3;

/* Set the remaining gas sensor settings and link the heating profile */
gas_sensor.gas_sett.run_gas = BME680_ENABLE_GAS_MEAS;
/* Create a ramp heat waveform in 3 steps */
gas_sensor.gas_sett.heatr_temp = 320; /* degree Celsius */
gas_sensor.gas_sett.heatr_dur = 150; /* milliseconds */

/* Select the power mode */
/* Must be set before writing the sensor configuration */
gas_sensor.power_mode = BME680_FORCED_MODE;

/* Set the required sensor settings needed */
set_required_settings = BME680_OST_SEL | BME680_OSP_SEL | BME680_OSH_SEL | BME680_FILTER_SEL
| BME680_GAS_SENSOR_SEL;

/* Set the desired sensor configuration */
rslt = bme680_set_sensor_settings(set_required_settings,&gas_sensor);

/* Set the power mode */
rslt = bme680_set_sensor_mode(&gas_sensor);

/* Get the total measurement duration so as to sleep or wait till the
 * measurement is complete */
uint16_t meas_period;
bme680_get_profile_dur(&meas_period, &gas_sensor);
user_delay_ms(meas_period); /* Delay till the measurement is ready */
```

Reading sensor data

Example for reading all sensor data

```
struct bme680_field_data data;

while(1)
{
    rslt = bme680_get_sensor_data(&data, &gas_sensor);

    printf("T: %.2f degC, P: %.2f hPa, H %.2f %%rH ", data.temperature / 100.0f,
           data.pressure / 100.0f, data.humidity / 1000.0f );
    /* Avoid using measurements from an unstable heating setup */
    if(data.status & BME680_GASM_VALID_MSK)
        printf(", G: %d ohms", data.gas_resistance);

    printf("\r\n");
}
```

Templates for function pointers

```
void user_delay_ms(uint32_t period)
{
    /*
     * Return control or wait,
     * for a period amount of milliseconds
     */
}

int8_t user_spi_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rs1t = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     * to be set low to activate the relevant device on the SPI bus
     */

    /*
     * Data on the bus should be like
     * |-----+-----+-----|
     * | MOSI      | MISO      | Chip Select |
     * |-----+-----+-----|
     * | (don't care) | (don't care) | HIGH |
     * | (reg_addr)   | (don't care) | LOW  |
     * | (don't care) | (reg_data[0]) | LOW  |
     * | (....)       | (....)       | LOW  |
     * | (don't care) | (reg_data[len - 1]) | LOW  |
     * | (don't care) | (don't care) | HIGH |
     * |-----+-----+-----|
     */
    return rs1t;
}

int8_t user_spi_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rs1t = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     * to be set low to activate the relevant device on the SPI bus
     */

    /*
     * Data on the bus should be like
     * |-----+-----+-----|
     * | MOSI      | MISO      | Chip Select |
     * |-----+-----+-----|
     * | (don't care) | (don't care) | HIGH |
     * | (reg_addr)   | (don't care) | LOW  |
     * | (reg_data[0]) | (don't care) | LOW  |
     * | (....)       | (....)       | LOW  |
     * | (reg_data[len - 1]) | (don't care) | LOW  |
     * | (don't care) | (don't care) | HIGH |
     * |-----+-----+-----|
     */
    return rs1t;
}

int8_t user_i2c_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rs1t = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to store the I2C address of the device
     */
}
```

```

* Data on the bus should be like
* |-----+-----|
* | I2C action | Data
* |-----+-----|
* | Start      | -
* | Write      | (reg_addr)
* | Stop       | -
* | Start      | -
* | Read       | (reg_data[0])
* | Read       | (....)
* | Read       | (reg_data[len - 1])
* | Stop       | -
* |-----+-----|
*/
return rs1t;
}

int8_t user_i2c_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rs1t = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to store the I2C address of the device
     */

    /*
     * Data on the bus should be like
     * |-----+-----|
     * | I2C action | Data
     * |-----+-----|
     * | Start      | -
     * | Write      | (reg_addr)
     * | Write      | (reg_data[0])
     * | Write      | (....)
     * | Write      | (reg_data[len - 1])
     * | Stop       | -
     * |-----+-----|
     */
    return rs1t;
}

```

Copyright (C) 2017 - 2018 Bosch Sensortec GmbH# Change Log

All notable changes to the BME680 Sensor API will be documented in this file.

v3.5.3, 30 Oct 2017

Changed

- Changed the compensation equation formulae to use shifting operation
- Updated the "bme680_get_profile_dur" API
- Fixed Checkpatch and made linux compatible

v3.5.2, 18 Oct 2017

Changed

- Fixed bug of temperature compensation in pressure

v3.5.1, 5 Jul 2017

Changed

- Fixed bug with overwriting of the result with communication results
- Added member in the dev structure to store communication results
- Updated set profile duration API to not return a result.
- Added new API to get the duration for the existing profile
- Fixed bug with setting gas configuration. Reduced to writing only relevant bytes
- Updated readme
- Updated documentation for the type definitions
- Removed mode check for get sensor data and setting and getting profile dur

v3.5.0, 28 Jun 2017

Changed

- Fixed bug with getting and setting mem pages
- Changed initialization sequence to be more robust
- Added additional tries while reading data in case of inadequate delay

v3.4.0, 8 Jun 2017

Changed

- Modified the bme680_get_sensor_data API. User has to now pass the struct that stores the data rather than retrieving from the bme680_dev structure.
- Fixed possible bugs

v3.3.0, 24 May 2017

Changed

- Name changes in the BME680 device structure.
- Removed sequential and parallel modes.
- Removed ODR related sensor settings
- Modified get sensor settings API with user selection.
- Removed sort sensor data and swap fields API which are not required.

Added

- BME680 set profile duration API.

v3.2.1, 17 May 2017

Added

- Took the reference as base version 3.2.1 of BME680 sensor and added.

BME680 sensor API

Introduction

This package contains the Bosch Sensortec's BME680 gas sensor API

The sensor driver package includes bme680.h, bme680.c and bme680_defs.h files

Version

File	Version	Date
bme680.c	3.5.9	19 Jun 2018
bme680.h	3.5.9	19 Jun 2018
bme680_defs.h	3.5.9	19 Jun 2018

Integration details

- Integrate bme680.h, bme680_defs.h and bme680.c file in to your project.
- Include the bme680.h file in your code like below.

```
#include "bme680.h"
```

File information

- bme680_defs.h : This header file has the constants, macros and datatype declarations.
- bme680.h : This header file contains the declarations of the sensor driver APIs.
- bme680.c : This source file contains the definitions of the sensor driver APIs.

Supported sensor interfaces

- SPI 4-wire
- I2C

Usage guide

Initializing the sensor

To initialize the sensor, you will first need to create a device structure. You can do this by creating an instance of the structure bme680_dev. Then go on to fill in the various parameters as shown below

Example for SPI 4-Wire

```

struct bme680_dev gas_sensor;

/* You may assign a chip select identifier to be handled later */
gas_sensor.dev_id = 0;
gas_sensor.intf = BME680_SPI_INTF;
gas_sensor.read = user_spi_read;
gas_sensor.write = user_spi_write;
gas_sensor.delay_ms = user_delay_ms;
/* amb_temp can be set to 25 prior to configuring the gas sensor
 * or by performing a few temperature readings without operating the gas sensor.
 */
gas_sensor.amb_temp = 25;

int8_t rslt = BME680_OK;
rslt = bme680_init(&gas_sensor);

```

Example for I2C

```

struct bme680_dev gas_sensor;

gas_sensor.dev_id = BME680_I2C_ADDR_PRIMARY;
gas_sensor.intf = BME680_I2C_INTF;
gas_sensor.read = user_i2c_read;
gas_sensor.write = user_i2c_write;
gas_sensor.delay_ms = user_delay_ms;
/* amb_temp can be set to 25 prior to configuring the gas sensor
 * or by performing a few temperature readings without operating the gas sensor.
 */
gas_sensor.amb_temp = 25;

int8_t rslt = BME680_OK;
rslt = bme680_init(&gas_sensor);

```

Regarding compensation functions for temperature, pressure, humidity and gas we have two implementations.

- Integer version
- floating point version

By default, Integer version is used in the API

If the user needs the floating point version, the user has to un-comment BME680_FLOAT_POINT_COMPENSATION macro in bme680_defs.h file or to add it in the compiler flags.

Configuring the sensor

Example for configuring the sensor in forced mode

```

uint8_t set_required_settings;

/* Set the temperature, pressure and humidity settings */
gas_sensor.tph_sett.os_hum = BME680_OS_2X;
gas_sensor.tph_sett.os_pres = BME680_OS_4X;
gas_sensor.tph_sett.os_temp = BME680_OS_8X;
gas_sensor.tph_sett.filter = BME680_FILTER_SIZE_3;

/* Set the remaining gas sensor settings and link the heating profile */
gas_sensor.gas_sett.run_gas = BME680_ENABLE_GAS_MEAS;
/* Create a ramp heat waveform in 3 steps */
gas_sensor.gas_sett.heatr_temp = 320; /* degree Celsius */
gas_sensor.gas_sett.heatr_dur = 150; /* milliseconds */

/* Select the power mode */
/* Must be set before writing the sensor configuration */
gas_sensor.power_mode = BME680_FORCED_MODE;

/* Set the required sensor settings needed */
set_required_settings = BME680_OST_SEL | BME680_OSP_SEL | BME680_OSH_SEL | BME680_FILTER_SEL
| BME680_GAS_SENSOR_SEL;

/* Set the desired sensor configuration */
rslt = bme680_set_sensor_settings(set_required_settings,&gas_sensor);

/* Set the power mode */
rslt = bme680_set_sensor_mode(&gas_sensor);

```

Reading sensor data

Example for reading all sensor data

```

/* Get the total measurement duration so as to sleep or wait till the
 * measurement is complete */
uint16_t meas_period;
bme680_get_profile_dur(&meas_period, &gas_sensor);

struct bme680_field_data data;

while(1)
{
    user_delay_ms(meas_period); /* Delay till the measurement is ready */

    rslt = bme680_get_sensor_data(&data, &gas_sensor);

    printf("T: %.2f degC, P: %.2f hPa, H %.2f %%rH ", data.temperature / 100.0f,
           data.pressure / 100.0f, data.humidity / 1000.0f );
    /* Avoid using measurements from an unstable heating setup */
    if(data.status & BME680_GASM_VALID_MSK)
        printf(", G: %d ohms", data.gas_resistance);

    printf("\r\n");

    /* Trigger the next measurement if you would like to read data out continuously */
    if (gas_sensor.power_mode == BME680_FORCED_MODE) {
        rslt = bme680_set_sensor_mode(&gas_sensor);
    }
}

```

Templates for function pointers

```

void user_delay_ms(uint32_t period)
{
    /*
     * Return control or wait,

```

```

* for a period amount of milliseconds
*/
}

int8_t user_spi_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     * to be set low to activate the relevant device on the SPI bus
     */

    /*
     * Data on the bus should be like
     * |-----+-----+-----|
     * | MOSI      | MISO      | Chip Select |
     * |-----+-----+-----|
     * | (don't care) | (don't care) | HIGH |
     * | (reg_addr)   | (don't care) | LOW  |
     * | (don't care) | (reg_data[0]) | LOW  |
     * | (....)       | (....)       | LOW  |
     * | (don't care) | (reg_data[len - 1]) | LOW  |
     * | (don't care) | (don't care) | HIGH |
     * |-----+-----+-----|
     */
}

return rslt;
}

int8_t user_spi_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     * to be set low to activate the relevant device on the SPI bus
     */

    /*
     * Data on the bus should be like
     * |-----+-----+-----|
     * | MOSI      | MISO      | Chip Select |
     * |-----+-----+-----|
     * | (don't care) | (don't care) | HIGH |
     * | (reg_addr)   | (don't care) | LOW  |
     * | (reg_data[0]) | (don't care) | LOW  |
     * | (....)       | (....)       | LOW  |
     * | (reg_data[len - 1]) | (don't care) | LOW  |
     * | (don't care) | (don't care) | HIGH |
     * |-----+-----+-----|
     */
}

return rslt;
}

int8_t user_i2c_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to store the I2C address of the device
     */

    /*
     * Data on the bus should be like
     * |-----+-----|
     * | I2C action | Data |
     * |-----+-----|
     * | Start      | -   |
     * | Write       | (reg_addr) |
     * | Stop        | -   |
     */
}

```

```

* | Start      | -
* | Read       | (reg_data[0])
* | Read       | (....)
* | Read       | (reg_data[len - 1])
* | Stop       | -
* |-----+-----|
*/
}

return rs1t;
}

int8_t user_i2c_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rs1t = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to store the I2C address of the device
     */

    /*
     * Data on the bus should be like
     * |-----+-----|
     * | I2C action | Data
     * |-----+-----|
     * | Start      | -
     * | Write      | (reg_addr)
     * | Write      | (reg_data[0])
     * | Write      | (....)
     * | Write      | (reg_data[len - 1])
     * | Stop       | -
     * |-----+-----|
     */
}

return rs1t;
}

```

**Copyright (C) 2017 - 2018 Bosch Sensortec GmbH Parts of the library
are covered by different licenses**

BME680 driver

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of the copyright holder nor the names of the contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE

The information provided is believed to be accurate and reliable. The copyright holder assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of the copyright holder.

BSEC library

```
/*
 * Copyright (C) 2015, 2016, 2017 Robert Bosch. All Rights Reserved..
 *
 * Disclaimer
 *
 * Common:
 * Bosch Sensortec products are developed for the consumer goods industry. They may only be used
 * within the parameters of the respective valid product data sheet. Bosch Sensortec products are
 * provided with the express understanding that there is no warranty of fitness for a particular purpose.
 * They are not fit for use in life-sustaining, safety or security sensitive systems or any system or device
 * that may lead to bodily harm or property damage if the system or device malfunctions. In addition,
 * Bosch Sensortec products are not fit for use in products which interact with motor vehicle systems.
 * The resale and/or use of products are at the purchasers own risk and his own responsibility. The
 * examination of fitness for the intended use is the sole responsibility of the Purchaser.
 *
 * The purchaser shall indemnify Bosch Sensortec from all third party claims, including any claims for
 * incidental, or consequential damages, arising from any product use not covered by the parameters of
 * the respective valid product data sheet or not approved by Bosch Sensortec and reimburse Bosch
 * Sensortec for all costs in connection with such claims.
 *
 * The purchaser must monitor the market for the purchased products, particularly with regard to
 * product safety and inform Bosch Sensortec without delay of all security relevant incidents.
 *
 * Engineering Samples are marked with an asterisk (*) or (e). Samples may vary from the valid
 * technical specifications of the product series. They are therefore not intended or fit for resale to third
 * parties or for use in end products. Their sole purpose is internal client testing. The testing of an
 * engineering sample may in no way replace the testing of a product series. Bosch Sensortec
 * assumes no liability for the use of engineering samples. By accepting the engineering samples, the
 * Purchaser agrees to indemnify Bosch Sensortec from all claims arising from the use of engineering
 * samples.
 *
 * Special:
 * This software module (hereinafter called "Software") and any information on application-sheets
 * (hereinafter called "Information") is provided free of charge for the sole purpose to support your
 * application work. The Software and Information is subject to the following terms and conditions:
 *
 * The Software is specifically designed for the exclusive use for Bosch Sensortec products by
 * personnel who have special experience and training. Do not use this Software if you do not have the
 * proper experience or training.
 *
 * This Software package is provided `` as is `` and without any expressed or implied warranties,
 * including without limitation, the implied warranties of merchantability and fitness for a particular
 * purpose.
 *
 * Bosch Sensortec and their representatives and agents deny any liability for the functional impairment
 * of this Software in terms of fitness, performance and safety. Bosch Sensortec and their
 * representatives and agents shall not be liable for any direct or indirect damages or injury, except as
 * otherwise stipulated in mandatory applicable law.
 *
 * The Information provided is believed to be accurate and reliable. Bosch Sensortec assumes no
 * responsibility for the consequences of use of such Information nor for any infringement of patents or
 * other rights of third parties which may result from its use. No license is granted by implication or
 * otherwise under any patent or patent rights of Bosch. Specifications mentioned in the Information are
 * subject to change without notice.
 *
 * It is not allowed to deliver the source code of the Software to any third party without permission of
 * Bosch Sensortec.
 *
 */

```

mOS integration code

Copyright (c) 2019 Deomid "rojer" Ryabkov
All rights reserved

Licensed under the Apache License, Version 2.0(the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

```# BME680 sensor driver library for Mongoose OS

## ## Overview

<p class="mume-header" id="overview-23"></p>

This library integrates the low-level [driver]([https://github.com/BoschSensortec/BME680\\_driver](https://github.com/BoschSensortec/BME680_driver)) and the [BSEC]([https://www.bosch-sensortec.com/bst/products/all\\_products/bsec](https://www.bosch-sensortec.com/bst/products/all_products/bsec)) software library which provides high-level sensor control to obtain reliable air quality sensor data.

This library provides the necessary glue code for both to work under Mongoose OS and a number of helper functions.

When sensor output is ready, an `MGOS\_EV\_BME680\_BSEC\_OUTPUT` event is triggered which receives a structure containing sensor outputs (see `struct mgos\_bme680\_t`).

## ### Platform support

<p class="mume-header" id="platform-support"></p>

Currently only supported on ESP8266 and ESP32 platforms, ARM support is a 'TODO'.

## ## Quick Start

<p class="mume-header" id="quick-start"></p>

The library is configured through the `bme680` configuration section, defined [here](mos.yml).

Several things need to be done to obtain readings from the sensor:

- I2C must be enabled, set `i2c.enable=true` (currently only I2C interface is supported).
- `bme680.i2c\_addr` must be set to the correct address. It's either 0x76 or 0x77 depending on the state of the address selection pin.
- `bme680.enable` needs to be set to `true` for library to be initialized at all.

With these and the rest of the settings left in their default state, you should get readings from all the sensors at 3 second interval.

## ## IAQ sensor accuracy

<p class="mume-header" id="iaq-sensor-accuracy"></p>

IAQ sensor requires calibration before producing accurate values. Values with accuracy value less than 3 are unreliable.

By default the library will perform calibration automatically (still may take up to 30 minutes to complete).

## ## Configuration details

<p class="mume-header" id="configuration-details"></p>

A number of options are provided for more advanced control of the sensor behavior.

- `bme680.bsec.enable`: normally it is advisable to use the BSEC library to process raw values returned by the sensor.

Turning this off will enable you to either use the sensor directly (reference to the dev ice can be obtained via `mgos\_bme680\_get\_global()`) or initialize driver directly.

- `bme680.bsec.config\_file`: BSEC library comes with a number of pre-generated configuration profiles that can be loaded to improve accuracy of the measurements.
- `bme680.bsec.state\_file`, `bme680.bsec.state\_save\_interval`: BSEC library performs estimations over long periods of time and the accuracy of its output relies on these parameters.
- `bme680.bsec.{iaq,temp,rh,ps}\_sample\_rate`: Set sampling rates for different parts of the BME680 multi-sensor. Each can be individually disabled (empty array).
- `bme680.bsec.iaq\_auto\_cal`: if IAQ sensor is enabled (`bme680.bsec.iaq\_sample\_rate` is not empty) and this option is enabled, mos will automatically raise an event when calibration is complete.

## ## Example

<p class="mume-header" id="example-4"></p>

With mos library providing the integration, getting samples from the sensor is very simple - all you need to do is subscribe to the event:

```
```c
#include "mgos.h"
#include "mgos_bme680.h"
```

```

static void bme680_output_cb(int ev, void *ev_data, void *arg) {
    const struct mgos_bsec_output *out = (struct mgos_bsec_output *) ev_data;
    double ts = out->temp.time_stamp / 1000000000.0;
    float ps_kpa = out->ps.signal / 1000.0f;
    float ps_mmhg = out->ps.signal / 133.322f;
    if (out->iaq.time_stamp > 0) {
        LOG(LL_INFO,
            ("%.2f IAQ %.2f (acc %d) T %.2f RH %.2f P %.2f kPa (%.2f mmHg)", ts,
            out->iaq.signal, out->iaq.accuracy, out->temp.signal, out->rh.signal,
            ps_kpa, ps_mmhg));
    } else {
        LOG(LL_INFO, ("%.2f T %.2f RH %.2f P %.2f kPa (%.2f mmHg)", ts,
            out->temp.signal, out->rh.signal, ps_kpa, ps_mmhg));
    }
}

(void) ev;
(void) arg;
}

enum mgos_app_init_result mgos_app_init(void) {
    mgos_event_add_handler(MGOS_EV_BME680_BSEC_OUTPUT, bme680_output_cb, NULL);
    return MGOS_APP_INIT_SUCCESS;
}

```

Output:

```

[Aug 26 23:00:59.324] mgos_i2c_gpio_maste:250 I2C GPIO init ok (SDA: 4, SCL: 5, freq: 100000)
[Aug 26 23:00:59.348] mgos_bme680.c:466      BME680 @ 0/0x77 init ok
[Aug 26 23:00:59.353] mgos_bme680.c:396      BSEC 1.4.7.4 initialized
[Aug 26 23:00:59.364] mgos_bme680.c:404      Failed to load BSEC config from bsec_iaq.config: -33, will use defaults
[Aug 26 23:00:59.377] mgos_bme680.c:414      Failed to load BSEC state from bsec.state: -33, will use defaults
...
[Aug 26 23:01:00.337] mgos_init.c:36      Init done, RAM: 51152 total, 41996 free, 42000 min free
[Aug 26 23:01:00.352] mgos_bme680.c:281      IAQ sensor requires calibration
[Aug 26 23:01:00.356] main.c:13      0.68 IAQ 25.00 (acc 0) T 27.07 RH 57.51 P 101.76 kPa (763.28 mmHg)
[Aug 26 23:01:02.641] main.c:13      3.68 IAQ 25.00 (acc 0) T 26.98 RH 57.87 P 101.76 kPa (763.29 mmHg)
[Aug 26 23:01:05.645] main.c:13      6.69 IAQ 25.00 (acc 0) T 26.99 RH 57.88 P 101.76 kPa (763.28 mmHg)
[Aug 26 23:01:08.650] main.c:13      9.69 IAQ 25.00 (acc 0) T 27.01 RH 57.84 P 101.76 kPa (763.28 mmHg)
...
[Aug 26 23:05:54.095] main.c:13      295.11 IAQ 25.00 (acc 0) T 26.72 RH 58.46 P 101.76 kPa (763.27 mmHg)
[Aug 26 23:05:56.868] mgos_bme680.c:368      BSEC state saved (bsec.state)
[Aug 26 23:05:57.100] main.c:13      298.11 IAQ 25.00 (acc 0) T 26.72 RH 58.46 P 101.76 kPa (763.26 mmHg)
...
[Aug 26 23:08:12.326] main.c:13      433.32 IAQ 51.91 (acc 1) T 26.70 RH 58.46 P 101.76 kPa (763.25 mmHg)
[Aug 26 23:08:15.331] main.c:13      436.33 IAQ 250.00 (acc 2) T 26.71 RH 58.41 P 101.76 kPa (763.27 mmHg)
...
[Aug 26 23:16:46.136] main.c:13      947.09 IAQ 49.84 (acc 3) T 26.72 RH 58.90 P 101.76 kPa (763.23 mmHg)
[Aug 26 23:16:49.140] mgos_bme680.c:289      IAQ sensor calibration complete
[Aug 26 23:16:49.144] main.c:13      950.10 IAQ 50.42 (acc 3) T 26.72 RH 58.92 P 101.76 kPa (763.23 mmHg)
[Aug 26 23:16:52.145] main.c:13      953.10 IAQ 50.01 (acc 3) T 26.73 RH 58.88 P 101.76 kPa (763.23 mmHg)
[Aug 26 23:16:55.150] main.c:13      956.10 IAQ 50.95 (acc 3) T 26.73 RH 58.87 P 101.76 kPa (763.23 mmHg)
...

```

License

See [here](#).

A VFS device that encrypts reads and writes

Overview

- AES-128/192/256 are supported (algo: AES-nnn parameter, default is AES-128).

- Reads and writes are encrypted, erases are passed through as is.
- Encryption is performed in ECB mode, key is XORed with offset.
- Reads and writes must be aligned to 16-byte boundaries.
- Writes will be padded to 16 byte block size, so partial writes will only work for last plain-text block.

Hint: If you want an encrypted filesystem, [LFS](#) will work just fine with this method while [SPIFFS](#) will not.

Key source

Key can be supplied directly (as the `key` option) but a better approach is to use a *key device* to obtain the key when required.

Key device can be any other VFS device that supports reads. It can be an existing device (`key_dev: name`) or created in-situ (`key_dev_type + key_dev_opts`).

Hint: To read key from RAM, use the `vfs-dev-ram`.

Hint 2: Want to generate your own key? Create your own VFS device. Don't worry about methods other than `read`.

Example

Options for encrypting `extf0` with AES-256 with key from STM32 OTP area (536836096 = 0x1fff7800).

```
{"dev": "extf0", "algo": "AES-256", "key_dev_type": "RAM", "key_dev_opts": {"addr": 536836096, "size": 32}}
```

Don't forget to add `vfs-dev-ram` to libs.

DNS-SD/Bonjour protocol support library

By default, advertises HTTP server (if enabled).

Additional instances can be added by user's application, example (for Apple HAP):

```
const struct mgos_dns_sd_txt_entry gizmo_txt[] = {
    {.key = "c#", .value = MG_MK_STR("1")},
    {.key = "ff", .value = MG_MK_STR("0")},
    {.key = "pv", .value = MG_MK_STR("1.0")},
    {.key = "id", .value = MG_MK_STR("11:22:33:44:55:66")},
    {.key = "md", .value = MG_MK_STR("Fancy Gizmo 9000")},
    {.key = "s#", .value = MG_MK_STR("1")},
    {.key = "sf", .value = MG_MK_STR("1")},
    {.key = "ci", .value = MG_MK_STR("8")}, // Switch
    {.key = NULL},
};

mgos_dns_sd_add_service_instance("gizmo9000", "_hap._tcp", 8080, gizmo_txt);
```

Cesanta note

Imported from https://github.com/adafruit/Adafruit_BME280_Library/commit/a148538212b07df05881562874171de7f312c81f

This is a library for the Adafruit BME280 Humidity, Barometric Pressure + Temp sensor

Designed specifically to work with the Adafruit BME280 Breakout

- <http://www.adafruit.com/products/2652>

These sensors use I2C or SPI to communicate, up to 4 pins are required to interface

Use of this library also requires [Adafruit_Sensor](#)

to be installed on your local system.

Adafruit invests time and resources providing this open source code,
please support Adafruit and open-source hardware by purchasing
products from Adafruit!

Check out the links above for our tutorials and wiring diagrams

Written by Limor Fried/Ladyada for Adafruit Industries.

BSD license, all text above must be included in any redistribution

To download, click the DOWNLOAD ZIP button, rename the uncompressed folder Adafruit_BME280.

Check that the Adafruit_BME280 folder contains Adafruit_BME280.cpp and Adafruit_BME280.h

Place the Adafruit_BME280 library folder your arduinosketchfolder/libraries/ folder.

You may need to create the libraries subfolder if its your first library. Restart the IDE.

We also have a great tutorial on Arduino library installation at:

<http://learn.adafruit.com/adafruit-all-about-arduino-libraries-install-use>

Compatibility

MCU	Tested Works	Doesn't Work	Not Tested	Notes
Atmega328 @ 16MHz	X			
Atmega328 @ 12MHz	X			
Atmega32u4 @ 16MHz	X			Use SDA/SCL on pins D2 & D3
Atmega32u4 @ 8MHz	X			Use SDA/SCL on pins D2 & D3
ESP8266	X			I2C: just works, SPI: SDA/SCL default to pins 4 & 5 but any two pins can be assigned as SDA/SCL using Wire.begin(SDA,SCL)
ESP32	X			I2C: just works, SPI: SDA/SCL default to pins 4 & 5 but any two pins can be assigned as SDA/SCL using Wire.begin(SDA,SCL)
Atmega2560 @ 16MHz	X			Use SDA/SCL on pins 20 & 21
ATSAM3X8E	X			Use SDA/SCL on pins 20 & 21
ATSAM21D	X			
ATtiny85 @ 16MHz		X		
ATtiny85 @ 8MHz		X		
Intel Curie @ 32MHz			X	
STM32F2			X	

- ATmega328 @ 16MHz : Arduino UNO, Adafruit Pro Trinket 5V, Adafruit Metro 328, Adafruit Metro Mini

- ATmega328 @ 12MHz : Adafruit Pro Trinket 3V
- ATmega32u4 @ 16MHz : Arduino Leonardo, Arduino Micro, Arduino Yun, Teensy 2.0
- ATmega32u4 @ 8MHz : Adafruit Flora, Bluefruit Micro
- ESP8266 : Adafruit Huzzah
- ATmega2560 @ 16MHz : Arduino Mega
- ATSAM3X8E : Arduino Due
- ATSAM21D : Arduino Zero, M0 Pro
- ATTiny85 @ 16MHz : Adafruit Trinket 5V
- ATTiny85 @ 8MHz : Adafruit Gemma, Arduino Gemma, Adafruit Trinket 3V

Arduino Adafruit BME280 and BMP280 library for Mongoose OS

VFS interface for SPIFFS

A collection of libraries for demoing JavaScript code

Overview

This library is intended to be used for apps that implement device logic in JavaScript. It is essentially a collection of libraries that implement a wide set of functionalities - from hardware peripherals API to cloud integrations like AWS IoT, Google IoT Core, etc.

Arduino DallasTemperature library for Mongoose OS

How to contribute

We'd love to accept your patches and contributions to this project. There are just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

Prometheus Sensors

This is a library that implements a set of sensor types, reading their values periodically and exposing them to Prometheus using the [prometheus-metrics](#) library.

Supported Sensors

DHT

This is using [dht](#) library; each sensor reading takes approximately 32ms (28ms if there is no sensor on the GPIO in). They are placed on timers so as to minimize pre-emption of the main application. DHT sensors are polled in order based on `sensors.dht_gpio` pins and with a period of `sensors.dht_period` which defaults to 3 seconds.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="DHT"} 3.5
temperature{sensor="1",type="DHT"} 6.2
humidity{sensor="0",type="DHT"} 43.900002
humidity{sensor="1",type="DHT"} 35.599998
```

VEML6075

This is using `veml6075-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x10`, but configurable with `sensors.veml6075_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.veml6075_period` which defaults to 3 seconds, each sensor reading takes approximately 15ms.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
UV{band="UVA",sensor="0",type="VEML6075"} 0  
UV{band="UVB",sensor="0",type="VEML6075"} 2  
UVIndex{sensor="0",type="VEML6075"} 0.00125
```

BME280

This is using `bme280` library; one sensor is allowed based on I2C address on the bus, normally `0x10`, but configurable with `sensors.bme280_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.bme280_period` which defaults to 3 seconds, each sensor reading takes approximately 4ms. Note that there are several versions of this popular barometer chip: BME280 is fully fledged and includes a hygrometer (measuring the relative humidity), while BMP280 does not.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="BME280"} 18.4  
humidity{sensor="0",type="BME280"} 77.4  
pressure{sensor="0",type="BME280"} 96720.4
```

MCP9808

This is using `mcp9808-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x18`, but configurable with `sensors.mcp9808_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.mcp9808_period` which defaults to 3 seconds, each sensor reading takes approximately 3ms.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="MCP9808"} 18.6
```

Si7021

This is using `si7021-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x40`, but configurable with `sensors.si7021_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.si7021_period` which defaults to 3 seconds, each sensor reading takes approximately 54ms.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="SI7021"} 18.6
humidity{sensor="0",type="SI7021"} 56.4
```

SHT31

This is using `sht31-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x44`, but configurable with `sensors.sht31_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.sht31_period` which defaults to 3 seconds, each sensor reading takes approximately 18ms.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="SHT31"} 18.6
humidity{sensor="0",type="SHT31"} 56.4
```

HTU21D-F

This is using `htu21df-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x40`, but configurable with `sensors.htu21df_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.htu21df_period` which defaults to 3 seconds, each sensor reading takes approximately 105ms.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
temperature{sensor="0",type="HTU21DF"} 18.5
humidity{sensor="0",type="HTU21DF"} 55.8
```

CCS811

This is using `ccs811-i2c` library; one sensor is allowed based on I2C address on the bus, normally `0x5A`, but configurable with `sensors.ccs811_i2caddr` in `mos.yml`. The chip is polled with a period of `sensors.ccs811_period` which defaults to 3 seconds, each sensor reading takes approximately 5ms.

Please see the upstream [source](#) for more information on the driver.

Reported values (all types are gauges):

```
eco2{sensor="0",type="CCS811"} 7992
tvoc{sensor="0",type="CCS811"} 1156
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

Implementation of Mongoose OS OTA HTTP client

This library adds a device configuration section called `update`, where a device could be configured to poll a specified HTTP URL for a new app firmware.

Also, this library adds a C API to fetch a new firmware from the given URL and update programmatically.

Configuration section

The library adds the following object to the device configuration:

```
"update": {  
    "commit_timeout": 0,           // OTA commit timeout  
    "url": "",                  // HTTP URL to poll  
    "interval": 0,               // Polling interval  
    "extra_http_headers": "",    // Extra HTTP request headers  
    "ssl_ca_file": "ca.pem",     // TLS CA cert file  
    "ssl_client_cert_file": "",  // TLS cert file  
    "ssl_server_name": "",       // TLS server name  
    "enable_post": true  
}
```

Si7005 Sensor Driver for Mongoose OS

Overview

[Si7005](#) is a temperature and relative humidity sensor by Silicon Labs.

This library provides a driver for this device.

API documentation

See `include/si7005.h`.

Example

```

#include "mgos.h"

#include "si7005.h"

static void temp_timer_cb(void *arg) {
    float temp = si7005_read_temp();
    float rh = si7005_read_rh();
    LOG(LL_INFO, ("T %.2f RH %.2f", temp, rh));
    (void) arg;
}

enum mgos_app_init_result mgos_app_init(void) {
    if (si7005_probe()) {
        LOG(LL_INFO, ("Si7005 sensor found"));
        mgos_set_timer(1000, MGOS_TIMER_REPEAT, temp_timer_cb, NULL);
    } else {
        LOG(LL_WARN, ("Failed to init temp sensor"));
    }
    return MGOS_APP_INIT_SUCCESS;
}

```

Note: You need to make sure that I2C is enabled. This can be achieved by adding

```

config_schema:
  - ["i2c.enable", true]

```

to mos.yml. You may need to adjust SDA and SCL pins as well.

See [I2C](#) library for details.

How to contribute

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial

issue that you are planning to work on that bug or feature so that it can be assigned to you.

2. Follow the normal process of [forking](#)

the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.

3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.

4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.

5. Do your best to have [well-formed commit messages](#)

for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.

6. Finally, push the commits to your fork and submit a [pull request](#).

MCP9808 I2C Driver

A Mongoose library for Microchip's MCP9808 integrated circuit.

Sensor details

Microchip Technology Inc.'s MCP9808 digital temperature sensor converts temperatures between -20°C and +100°C to a digital word with $\pm 0.25^\circ\text{C}/\pm 0.5^\circ\text{C}$ (typical/maximum) accuracy.

The MCP9808 comes with user-programmable registers that provide flexibility for temperature sensing applications. The registers allow user-selectable settings such as Shutdown or Low-Power modes and the specification of temperature Alert window limits and critical output limits. When the temperature changes beyond the specified boundary limits, the MCP9808 outputs an Alert signal. The user has the option of setting the Alert output signal polarity as an active-low or activehigh comparator output for thermostat operation, or as a temperature Alert interrupt output for microprocessor based systems. The Alert output can also be configured as a critical temperature output only.

This sensor has an industry standard 400 kHz, 2-wire, SMBus/I2C compatible serial interface, allowing up to eight or sixteen sensors to be controlled with a single serial bus.

These features make the MCP9808 ideal for sophisticated, multi-zone, temperature-monitoring applications.

See [datasheet](#)

for implementation details.

A great place to buy a ready made and tested unit is at [Adafruit](#).

Example application

An example program using a timer to read data from the sensor every 5 seconds:

```

#include "mgos.h"
#include "mgos_i2c.h"
#include "mgos_mcp9808.h"

static struct mgos_mcp9808 *s_mcp9808;

static void timer_cb(void *user_data) {
    float temperature;

    temperature=mgos_mcp9808_getTemperature(s_mcp9808);

    LOG(LL_INFO, ("mcp9808 temperature=%.2f", temperature));

    (void) user_data;
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_i2c *i2c;

    i2c=mgos_i2c_get_global();
    if (!i2c) {
        LOG(LL_ERROR, ("I2C bus missing, set i2c.enable=true in mos.yml"));
    } else {
        s_mcp9808=mgos_mcp9808_create(i2c, 0x40); // Default I2C address
        if (s_mcp9808) {
            mgos_set_timer(5000, true, timer_cb, NULL);
        } else {
            LOG(LL_ERROR, ("Could not initialize sensor"));
        }
    }
    return MGOS_APP_INIT_SUCCESS;
}

```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

JSON Store

JSON Store is a library which maintains a JSON file with order-preserving mappings from arbitrary strings to any kind of JSON values (a string, an object, etc). IDs can be either randomly generated or custom.

The order in which items are stored in the file and iterated (with `mgos_jstore_iterate()`, see `include/mgos_jstore.h`) is the order of insertion of items to the store.

RPC over GATT, Server

Overview

This library provides a GATT service that acts as an RPC channel. It accepts incoming frames and can send them as well - or rather, make them available for collection.

Note: Default BT configuration is permissive. See <https://github.com/mongoose-os-libs/bt-common#security> for a better idea.

Attribute description

The service UUID is 5f6d4f53-5f52-5043-5f53-56435f49445f, which is a representation of a 16-byte string _mOS_RPC_SVC_ID_.

Three attributes are defined:

- 5f6d4f53-5f52-5043-5f64-6174615f5f (_mOS_RPC_data_) - a r/w attribute used to submit frames for tx to the device and read out frames from the device.
- 5f6d4f53-5f52-5043-5f74-785f63746c5f (_mOS_RPC_tx_ctl_) - a write-only attribute. Before sending a frame expected length of the frame is submitted as a big-endian 32-bit number (so, for a 100 byte frame bytes 00 00 00 64 should be sent), followed by any number of writes to the data attribute. Chunking can be arbitrary, but the result must add up to the specified length exactly, at which point frame will be processed. Write to tx_ctl clears out any half-written frame that might be buffered, so writer needs to ensure there's only one frame in flight at any time.
- 5f6d4f53-5f52-5043-5f72-785f63746c5f (_mOS_RPC_rx_ctl_) - a read/notify attribute. It returns the length of the frame that device wishes to transmit as a big-endian 32-bit number. If this value is not zero, frame data will be returned in response to read requests of the data attribute. Read chunks will be up to MTU bytes in size. Client may subscribe to notifications on this attribute. Notification will be sent whenever a new frame is submitted for delivery and the notification body will contain length (the same value as returned by reading). Upon receiving notification client can proceed to read the data without reading rx_ctl again.

Lightning How To

- Add - origin: <https://github.com/mongoose-os-libs/rpc-gatts> to your mos.yml libs
- For this example we'll be doing a wifi.scan as that's very handy! Add - origin: <https://github.com/mongoose-os-libs/rpc-service-wifi> to your mos.yml libs as well.
- Ensure your mos.yml config.schema has:

```
- ["bt.enable", "b", true, {title: "Enable the BT service"}]
- ["bt.keep_enabled", "b", true, {title: "Keep the BT service enabled despite WIFI being up"}]
- ["bt.random_address", "b", false, {title: "Use a consistent BT MAC address"}]
- ["bt.dev_name", "MyDeviceName_?????"]
```

- Rebuild/flash your app
- Run a BLE scanner (ie Bluesee) to see your MyDeviceName_????? device. Connect to see the services. Look for the one with 5f52-5043 in the middle.
- Select that service to open it. In Bluesee you'll have 3 characteristics per above docu.
- Figure out your RPC command, we'll use: {"id":1999,"method":"Wifi.Scan"}
- We need to tell the service ahead of time how many hex bytes we'll be sending. We need to convert ascii to hex, trim the extra spaces and then count the bytes. Fun!
- Enter your RPC command into this ascii to hex converter: <https://www.rapidtables.com/convert/number/ascii-to-hex.html>
- It'll give you a space separated hex output: 7b 22 69 64 22 3a 31 39 39 39 2c 22 6d 65 74 68 6f 64 22 3a 22 57 69 66 69 2e 53 63 6 1 6e 22 7d
- Trim the spaces with <http://www.unit-conversion.info/texttools/replace-text/>
- Count the new hex string with: <https://mothereff.in/byte-counter>
- Halve the string count, as two hex characters equal one byte of data. With the text above, 64 / 2 = 32.
- Convert 32 to the large 32 bit number the system is expecting. Note: It's little endian, not big endian. Use this converter: <https://www.scadacore.com/tools/programming-calculators/online-hex-converter/>. Put the 32 in the input, then copy/paste the 'UINT32 - Little Endian (DCBA)' result and trim spaces:00000020
- Enter that 00000020 into the write only characteristic that has 5043-5F74 in the middle. Press write. In the console you'll see a log entry like: mgos_rpc_channel_ga:296 0x3ffe1d24 expected_flen 32 res 0
- On the middle read only characteristic with 5043-5F72 in the middle, click subscribe. The number will change to let us know when data is available (and how long the hex string is)
- On the read/write characteristic with 5043-5F53 in the middle, toggle the hex button so it says Ascii
- Paste in our ascii command: {"id":1999,"method":"Wifi.Scan"}
- You'll see the middle read only characteristic will have changed to let us know data is ready
- Click 'read' on the top read/write characteristic we just entered the scan command into. It'll start pulling the data of the scan results!

You can click it multiple times to scroll through the data till the end.

- You can repeat the request by clicking the write only characteristic to send it our command length, then paste our command again, and click write.

Arduino SPI library for Mongoose OS

This library provides an Arduino compatibility layer for SPI by providing an SPI.h public header, so that one could pick an existing Arduino program which uses SPI, throw it into the Mongoose OS application sources, and ideally, it "just works".

AWS IoT support for Mongoose OS

This library implements integration of Mongoose OS with AWS IoT.

See tutorial at <https://mongoose-os.com/docs/mongoose-os/cloud/aws.md>

Mongoose Embedded Web Server Library

See [here](#).

BME280 sensor API

Introduction

This package contains the Bosch Sensortec's BME280 pressure sensor driver (sensor API)

The sensor driver package includes bme280.c, bme280.h and bme280_defs.h files.

Version

File	Version	Date
bme280.c	3.3.4	14 Feb 2018
bme280.h	3.3.4	14 Feb 2018
bme280_defs.h	3.3.4	14 Feb 2018

Integration details

- Integrate bme280.h, bme280_defs.h and bme280.c file in to the project.
- Include the bme280.h file in your code like below.

```
#include "bme280.h"
```

File information

- bme280_defs.h : This header file has the constants, macros and datatype declarations.

- bme280.h : This header file contains the declarations of the sensor driver APIs.
- bme280.c : This source file contains the definitions of the sensor driver APIs.

Supported sensor interfaces

- SPI 4-wire
- I2C

SPI 3-wire is currently not supported in the API.

Usage guide

Initializing the sensor

To initialize the sensor, user need to create a device structure. User can do this by creating an instance of the structure bme280_dev. After creating the device structure, user need to fill in the various parameters as shown below.

Example for SPI 4-Wire

```
struct bme280_dev dev;
int8_t rslt = BME280_OK;

/* Sensor_0 interface over SPI with native chip select line */
dev.dev_id = 0;
dev.intf = BME280_SPI_INTF;
dev.read = user_spi_read;
dev.write = user_spi_write;
dev.delay_ms = user_delay_ms;

rslt = bme280_init(&dev);
```

Example for I2C

```
struct bme280_dev dev;
int8_t rslt = BME280_OK;

dev.dev_id = BME280_I2C_ADDR_PRIM;
dev.intf = BME280_I2C_INTF;
dev.read = user_i2c_read;
dev.write = user_i2c_write;
dev.delay_ms = user_delay_ms;

rslt = bme280_init(&dev);
```

Regarding compensation functions for temperature,pressure and humidity we have two implementations.

1. Double precision floating point version
2. Integer version

By default, integer version is used in the API. If the user needs the floating point version, the user has to uncomment BME280_FLOAT_ENABLE macro in bme280_defs.h file or add that to the compiler flags.

In integer compensation functions, we also have below two implementations for pressure.

1. For 32 bit machine.
2. For 64 bit machine.

By default, 64 bit variant is used in the API. If the user wants 32 bit variant, the user can disable the macro BME280_64BIT_ENABLE in bme280_defs.h file.

Sensor data units

The sensor data units depends on the following macros being enabled or not,
(in bme280_defs.h file or as compiler macros)

- BME280_FLOAT_ENABLE
- BME280_64BIT_ENABLE

In case of the macro "BME280_FLOAT_ENABLE" enabled,

The outputs are in double and the units are

- °C for temperature
- % relative humidity
- Pascal for pressure

In case if "BME280_FLOAT_ENABLE" is not enabled, then it is

- int32_t for temperature with the units 100 * °C
- uint32_t for humidity with the units 1024 * % relative humidity
- uint32_t for pressure
 - If macro "BME280_64BIT_ENABLE" is enabled, which it is by default, the unit is 100 * Pascal
 - If this macro is disabled, Then the unit is in Pascal

Stream sensor data

Stream sensor data in forced mode

```
int8_t stream_sensor_data_forced_mode(struct bme280_dev *dev)
{
    int8_t rslt;
    uint8_t settings_sel;
    struct bme280_data comp_data;

    /* Recommended mode of operation: Indoor navigation */
    dev->settings.osr_h = BME280_OVERSAMPLING_1X;
    dev->settings.osr_p = BME280_OVERSAMPLING_16X;
    dev->settings.osr_t = BME280_OVERSAMPLING_2X;
    dev->settings.filter = BME280_FILTER_COEFF_16;

    settings_sel = BME280_OSR_PRESS_SEL | BME280_OSR_TEMP_SEL | BME280_OSR_HUM_SEL | BME280_FILTER_SEL;

    rslt = bme280_set_sensor_settings(settings_sel, dev);

    printf("Temperature, Pressure, Humidity\r\n");
    /* Continuously stream sensor data */
    while (1) {
        rslt = bme280_set_sensor_mode(BME280_FORCED_MODE, dev);
        /* Wait for the measurement to complete and print data @25Hz */
        dev->delay_ms(40);
        rslt = bme280_get_sensor_data(BME280_ALL, &comp_data, dev);
        print_sensor_data(&comp_data);
    }
    return rslt;
}

void print_sensor_data(struct bme280_data *comp_data)
{
#ifdef BME280_FLOAT_ENABLE
    printf("%0.2f, %0.2f, %0.2f\r\n", comp_data->temperature, comp_data->pressure, comp_data->humidity);
#else
    printf("%ld, %ld, %ld\r\n", comp_data->temperature, comp_data->pressure, comp_data->humidity);
#endif
}
```

Stream sensor data in normal mode

```

int8_t stream_sensor_data_normal_mode(struct bme280_dev *dev)
{
    int8_t rslt;
    uint8_t settings_sel;
    struct bme280_data comp_data;

    /* Recommended mode of operation: Indoor navigation */
    dev->settings.osr_h = BME280_OVERSAMPLING_1X;
    dev->settings.osr_p = BME280_OVERSAMPLING_16X;
    dev->settings.osr_t = BME280_OVERSAMPLING_2X;
    dev->settings.filter = BME280_FILTER_COEFF_16;
    dev->settings.standby_time = BME280_STANDBY_TIME_62_5_MS;

    settings_sel = BME280_OSР_PRESSEL;
    settings_sel |= BME280_OSР_TEMP_SEL;
    settings_sel |= BME280_OSР_HUM_SEL;
    settings_sel |= BME280_STANDBY_SEL;
    settings_sel |= BME280_FILTER_SEL;
    rslt = bme280_set_sensor_settings(settings_sel, dev);
    rslt = bme280_set_sensor_mode(BME280_NORMAL_MODE, dev);

    printf("Temperature, Pressure, Humidity\r\n");
    while (1) {
        /* Delay while the sensor completes a measurement */
        dev->delay_ms(70);
        rslt = bme280_get_sensor_data(BME280_ALL, &comp_data, dev);
        print_sensor_data(&comp_data);
    }

    return rslt;
}

void print_sensor_data(struct bme280_data *comp_data)
{
#ifdef BME280_FLOAT_ENABLE
    printf("%0.2f, %0.2f, %0.2f\r\n", comp_data->temperature, comp_data->pressure, comp_data->humidity);
#else
    printf("%ld, %ld, %ld\r\n", comp_data->temperature, comp_data->pressure, comp_data->humidity);
#endif
}

```

Templates for function pointers

```

void user_delay_ms(uint32_t period)
{
    /*
     * Return control or wait,
     * for a period amount of milliseconds
     */
}

int8_t user_spi_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     * to be set low to activate the relevant device on the SPI bus
     */

    /*
     * Data on the bus should be like
     * |-----+-----+-----|
     * | MOSI      | MISO      | Chip Select |
     * |-----+-----+-----|
     * | (don't care) | (don't care) | HIGH      |
     * | (reg_addr)   | (don't care) | LOW       |
     * | (don't care) | (reg_data[0]) | LOW       |
     * | (....)       | (....)     | LOW       |
     */
}
```

```

* | (don't care) | (reg_data[len - 1]) | LOW
* | (don't care) | (don't care) | HIGH
* | -----+-----+-----|
*/
}

return rslt;
}

int8_t user_spi_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to select which Chip Select pin has
     * to be set low to activate the relevant device on the SPI bus
     */
}

/*
 * Data on the bus should be like
* |-----+-----+-----|
* | MOSI      | MISO      | Chip Select |
* |-----+-----+-----|
* | (don't care) | (don't care) | HIGH
* | (reg_addr) | (don't care) | LOW
* | (reg_data[0]) | (don't care) | LOW
* | (....) | (....) | LOW
* | (reg_data[len - 1]) | (don't care) | LOW
* | (don't care) | (don't care) | HIGH
* |-----+-----+-----|
*/
}

return rslt;
}

int8_t user_i2c_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to store the I2C address of the device
     */
}

/*
 * Data on the bus should be like
* |-----+-----|
* | I2C action | Data |
* |-----+-----|
* | Start | -
* | Write | (reg_addr)
* | Stop | -
* | Start | -
* | Read | (reg_data[0])
* | Read | (....)
* | Read | (reg_data[len - 1])
* | Stop | -
* |-----+-----|
*/
}

return rslt;
}

int8_t user_i2c_write(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len)
{
    int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

    /*
     * The parameter dev_id can be used as a variable to store the I2C address of the device
     */
}

/*
 * Data on the bus should be like
* |-----+-----|

```

```

* | I2C action | Data
* |-----+-----|
* | Start      | -
* | Write       | (reg_addr)
* | Write       | (reg_data[0])
* | Write       | (....)
* | Write       | (reg_data[len - 1])
* | Stop        | -
* |-----+-----|
*/
return rslt;
}

```

Copyright (C) 2016 - 2017 Bosch Sensortec GmbH# Bosch Sensortec's BMP280/BME280 Mongoose OS library

It uses Bosch Sensortec's reference implementation.

Both BMP280 and BME280 are supported with I2C or 4-wire SPI interface.

The library uses global instances of I2C or SPI with `spi.cs0_gpio`.

If different pins than the default ones are used, the user should define them in the `config_schema` of `mos.yml`

LwIP TCP/IP stack

[LwIP](#) version 2.1.2 + FreeRTOS bindings.

This library is used by ports other than ESP8266 and ESP32 (which ship their own, highly modified versions).

Arduino Adafruit - 16-Channel 12-bit PWM/Servo Driver - I2C interface - PCA9685 library for Mongoose OS

Tested and works on esp8266/esp32

Description

16-Channel 12-bit PWM/Servo Driver – I2C interface – PCA9685 Module lets you add more IOs to your microcontroller boards. The Module has chainable I2C interface meaning for even more IOs more boards can be daisy chained. Its an excellent product for Robots that require lots of Servo motors to control using single microcontroller. See [product page link](#) for more information about the hardware.

Features

- It is 5V compliant, which means you can control it from a 3.3V microcontroller and still safely drive up to 6V outputs (this is good for when you want to control white or blue LEDs with 3.4+ forward voltages)
- 6 address select pins so you can wire up to 62 of these on a single i2c bus, a total of 992 outputs – that's a lot of servos or LEDs
- Adjustable frequency PWM up to about 1.6 KHz
- 12-bit resolution for each output – for servos, that means about 4us resolution at 60Hz update rate

- Configurable push-pull or open-drain output
- Output enable pin to quickly disable all the outputs
- Reverse polarity protection on the terminal block input
- Power-good LED
- 3 pin connectors in groups of 4 so you can plug in 16 servos at once
- A big capacitor on the V+ line
- 220 ohm series resistors on all the output lines to protect them, and to make driving LEDs trivial
- IO expander chip: PCA9685PW
- Operating Voltage: 2.3V – 5.5V
- Solder jumpers for the 6 address select pins
- I2C 7-bit address Range: between 0x40-0x7F, selectable with jumpers

Usage

mos.yml, add:

```
config_schema:
- ["i2c.enable", true]
libs:
- origin: https://github.com/mongoose-os-libs/arduino-adafruit-pwm-servo
```

init.js, add:

```
load('api_pwm_servo.js');
```

main.c, add:

```
# include "mgos_arduino_PWMservoDriver.h"
<p class="mume-header" id="include-mgos_arduino_pwmservodriverh"></p>
```

Slave address

- Board 0: Address = 0x40 Offset = binary 00000 (no jumpers required)
- Board 1: Address = 0x41 Offset = binary 00001 (bridge A0)
- Board 2: Address = 0x42 Offset = binary 00010 (bridge A1)
- Board 3: Address = 0x43 Offset = binary 00011 (bridge A0 & A1)
- ...

Crontab RPC Service

How to contribute

We'd love to accept your patches and contributions to this project. There are a just a few small guidelines you need to follow.

Contributor License Agreement

Contributions to any Google project must be accompanied by a Contributor License Agreement. This is not a copyright **assignment**, it simply gives Google permission to use and redistribute your contributions as part of the project.

If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](#).

If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](#).

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

Submitting a patch

1. It's generally best to start by opening a new issue describing the bug or feature you're intending to fix. Even if you think it's relatively minor, it's helpful to know what people are working on. Mention in the initial issue that you are planning to work on that bug or feature so that it can be assigned to you.
2. Follow the normal process of [forking](#) the project, and setup a new branch to work in. It's important that each group of changes be done in separate branches in order to ensure that a pull request only includes the commits related to that bug or feature.
3. Any significant changes should almost always be accompanied by tests. The project already has good test coverage, so look at some of the existing tests if you're unsure how to go about it.
4. All contributions must be licensed Apache 2.0 and all files must have a copy of the boilerplate licence comment which can be generated by the [autogen](#) tool.
5. Do your best to have [well-formed commit messages](#) for each change. This provides consistency throughout the project, and ensures that commit messages are able to be formatted properly by various git tools.
6. Finally, push the commits to your fork and submit a [pull request](#).

PCF8574/PCF8575 I2C Driver

A Mongoose library for PCF857X, a popular and cheap set of GPIO extenders using I2C.

The PCF8574 is an 8-port device, and the PCF8575 is a 16-port device, but they are otherwise identical.

Implementation details

The PCF857X is a pseudo bi-directional GPIO, which means that by default ports are input and output. Writes to the ports can change the state from high to low. If the port is high, it is pulled up by a weak current. Each port can be pulled low by an external device, in which case reads from that port return 0. The implication is that "input" ports are always pulled high.

API Description

To start, `mgos_pcf857x_create()` is called with the correct I2C bus and address (by default 0x20), and optionally a GPIO pin on the microcontroller that

serves as an interrupt pin, to detect PCF857X input state changes.

NOTE: When the driver starts, it polls the current state from the chip without changing any ports. The benefit of this is that the MCU can safely reboot without loss of the GPIO state in PCF857X.

The API follows `mgos_gpio.h` closely, enabling ports to be set as input (ignoring the `pull_type`, which is always pullup), or as output. Notably, `mgos_pcf857x_gpio_set_int_handler()` and `mgos_pcf857x_gpio_set_button_handler()` work identically to the `mgos_gpio_*`() variants.

Example application

```
# include "mgos.h"
<p class="mume-header " id="include-mgosh-33"></p>

# include "mgos_config.h"
<p class="mume-header " id="include-mgos_configh-7"></p>

# include "mgos_pcf857x.h"
<p class="mume-header " id="include-mgos_pcf857xh"></p>

static void button_cb(int pin, void *user_data) {
    struct mgos_pcf857x *d = (struct mgos_pcf857x *)user_data;
    LOG(LL_INFO, ("GPIO[%d] callback, value=%d", pin, mgos_pcf857x_gpio_read(d, pin)));
    mgos_pcf857x_gpio_toggle(d, pin+4);
}

enum mgos_app_init_result mgos_app_init(void) {
    struct mgos_pcf857x *d;
    int i;

    if (!(d = mgos_pcf8574_create(mgos_i2c_get_global(), mgos_sys_config_get_pcf857x_i2caddr(),
        mgos_sys_config_get_pcf857x_int_gpio())) {
        LOG(LL_ERROR, ("Could not create PCF857X"));
        return MGOS_APP_INIT_ERROR;
    }

    for(i=0; i<4; i++) mgos_pcf857x_gpio_setup_input(d, i, MGOS_GPIO_PULL_UP);
    for(i=4; i<8; i++) mgos_pcf857x_gpio_set_mode(d, i, MGOS_GPIO_MODE_OUTPUT);

    mgos_pcf857x_gpio_set_button_handler(d, 0, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_NEG, 10, button_cb, d);
    mgos_pcf857x_gpio_set_button_handler(d, 1, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_POS, 10, button_cb, d);
    mgos_pcf857x_gpio_set_button_handler(d, 2, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_ANY, 10, button_cb, d);
    mgos_pcf857x_gpio_set_button_handler(d, 3, MGOS_GPIO_PULL_UP, MGOS_GPIO_INT_EDGE_ANY, 10, button_cb, d);

    return MGOS_APP_INIT_SUCCESS;
}
```

Disclaimer

This project is not an official Google project. It is not supported by Google and Google specifically disclaims all warranties as to its quality, merchantability, or fitness for a particular purpose.

This library adds I2C master support for the Mongoose OS. See default pin mapping in the yml file for your platform: `mos_<PLATFORM>.yml`.

Arduino Adafruit ADS1015 12-Bit/ADS1115 16-Bit ADC - 4 Channel with Programmable Gain Amplifier library for Mongoose OS

Tested and works on esp8266/esp32

Description

Analog-to-digital converter or higher-precision ADC.

	precision	samples\sec
ADS1015	12-bit	3300
ADS1115	16-bit	860

The chip can be configured as 4 single-ended input channels, or two differential channels. As a nice bonus, it even includes a programmable gain amplifier, up to x16, to help boost up smaller single/differential signals to the full range. See [ADS1015](#) [ADS1115](#) for more information about the hardware.

Features

- Wide supply range: 2.0V to 5.5V
- Low current consumption: Continuous Mode: Only 150uA Single-Shot Mode: Auto Shut-Down
- Programmable data rate: 8SPS to 860SPS
- Internal Low-drift voltage reference
- Internal oscillator
- Internal PGA
- I2C interface: Pin-Selectable Addresses
- Four Single-ended or two differential inputs
- Programmable comparator
- This board/chip uses I2C 7-bit addresses between 0x48-0x4B, selectable with jumpers

Usage

mos.yml, add:

```
config_schema:  
- ["i2c.enable", true]  
libs:  
- origin: https://github.com/mongoose-os-libs/arduino-adafruit-ads1x15
```

init.js, add:

```
load('api_ads1015.js');
```

main.c, add:

```
#include "mgos_arduino_Adafruit_AMG88xx.h"
```

Slave address

- Address pin connected to GND = 0x48 Offset = binary 1001000
- Address pin connected to VDD = 0x49 Offset = binary 1001001
- Address pin connected to SDA = 0x4A Offset = binary 1001010
- Address pin connected to SCL = 0x4B Offset = binary 1001011

RPC Service - GPIO

This service provides an ability to manage GPIO on devices remotely.
It is possible to call this service programmatically via serial, HTTP/RESTful,
Websocket, MQTT or other transports
(see [RPC section](#)) or use `mos` tool.

Below is a list of exported RPC methods and arguments:

GPIO.Read

Set given pin in INPUT mode, read GPIO pin, return its value. Arguments:

```
{  
  "pin": 15      // Required. Pin number.  
}
```

Example usage:

```
mos call GPIO.Read '{"pin": 0}'  
{  
  "value": 1  
}
```

GPIO.Write

Set given pin in OUTPUT mode, set GPIO pin. Arguments:

```
{  
  "pin": 15,    // Required. Pin number.  
  "value": 0    // Required. Voltage level. Either 0 (low) or 1 (high).  
}
```

Example usage:

```
mos call GPIO.Write '{"pin": 2, "value": 0}'
```

GPIO.Toggle

Set given pin in OUTPUT mode, toggle voltage level and return that level. Arguments:

```
{  
    "pin": 15      // Required. Pin number.  
}
```

Example usage:

```
mos call GPIO.Toggle '{"pin": 2}'  
{  
    "value": 1  
}
```

Mongoose OS documentation

Mongoose OS is a firmware development framework for microcontrollers.

Mongoose OS was created for developers who work on commercial connected products, and care about things like manageability, security, and reliability.

80-90% of the commercial firmware is around infrastructure, which is the same for many products regardless of what they do. Mongoose OS provides such a reliable, field-tested infrastructure.

Quick Summary

- Designed for commercial products
- Reliable OTA
- Support for major cloud services: AWS IoT, Microsoft Azure, Google IoT Core, IBM Watson, Samsung Artik
- Support for generic in-house MQTT, RESTful, Websocket servers
- Networking core is based on the mature [Mongoose Networking Library](#)
- RPC infrastructure for remote management:
 - JSON-RPC 2.0 framing
 - transports: MQTT, UART, BLE, REST, Websocket
 - many built-in services like Config, FS, GPIO, I2C, GATTC, Wifi, etc
- Configuration infrastructure with "reset to factory defaults"
- Advanced security features
 - ECC508A crypto element support
 - TLS 1.2 based on the ARM mbedTLS
 - low footprint tuning of the TLS stack
- Modular: [small core](#) and [over a hundred libraries](#)
- Commercial friendly Apache 2.0 license

Smart Light IoT product reference

We have published an end-to-end IoT product reference, <https://github.com/cesanta/mongoose-os-smart-light>. It could be used as a baseline for your own commercial product! See usage guide video:

Other resources

- [Developer forum](#)
- [Developer chat](#)
- [Youtube video tutorials](#)

This documentation is browseable online at <https://mongoose-os.com/docs>

UART gateway

Introduction

UART gateway connects any device to the Internet via UART.

Everything that your device sends to the UART, the gateway forwards to the Internet server - for example, AWS IoT, or Azure, or your own private TCP / Websocket backend.

Likewise, everything that is sent to the gateway, gateway forwards to the UART.

The UART gateway could report to the BLE central, to the raw TCP server, to the Websocket server, and to the MQTT server. Also, it can accept an incoming TCP/Websocket connections. Thus it is very versatile in terms of the connection options.

UART gateway does not implement any logic whilst forwarding data to and from the UART. Data is transferred as-is, no modification happens in any direction. Message boundaries are determined by timing.

Authentication and encryption is provided by the TLS layer.

The server side must be a TLS1.2 server with client-side authentication (mutual TLS) for the industry-standard level of security. AWS IoT is an example of such service.

Quick start

1. Buy hardware mentioned on the [Uart gateway page](#)
2. Connect hardware to your workstation via USB
3. Follow steps 1,2,3 of [mos setup](#)
4. In the `mos` UI, run `mos flash https://mongoose-os.com/downloads/uart-gateway/uart-gateway.zip` command
5. Follow step 7 of [mos setup](#)

6. Login to the [license manager](#) and buy uart-gateway licenses (one per device)
7. Execute `mos license`
8. Configure MQTT:
 - For generic/private MQTT server, run

```
mos config-set mqtt.enable=true mqtt.server=HOST:PORT tu.mqtt.enable=true
```

 - For AWS, follow [AWS guide's](#) "Setup AWS IoT" and "Setup device" chapters
 - For Google, follow [GCP guide's](#) "Setup Google IoT Core" and "Setup device" chapters
 - For Azure, follow [Azure guide's](#) "Setup Azure IoT Hub" and "Setup device" chapters
 - For Watson, follow [Watson guide's](#) "Quick setup" chapter
 - For mDash, follow step 8 of [mos setup](#)
9. Connect UART: pin 25 to TX, pin 26 to RX

When done, you should have your device flashed, provisioned to WiFi, connected to the cloud, and reporting UART data. Below is an example of the eval ESP32 Devkit-C board that connects a USB-to-Serial, making computer's serial device controlled via MQTT:

USB-to-Serial converter	UART gateway on ESP32 Devkit-C
GND	GND
RX	IO25
TX	IO26

reporting data to the <http://www.mqtt-dashboard.com>:

Configuring UART gateway

UART gateway keeps its configuration on a flash filesystem. It could be instected and changed using `mos` tool. To see an existing configuration, execute `mos config-get`. Below is the documentation for relevant entries:

```

"tu": {
  "bt": { // BLE settings
    "rx_buf_size": 2048, // BLE RX buffer size
    "tx_buf_size": 2048 // BLE TX buffer size
  },
  "evict_old": true,
  "mqtt": { // MQTT settings
    "enable": true, // Enable MQTT
    "qos": 0, // MQTT publishing QoS
    "rx_topic": "uart-bridge/rx", // RX topic
    "tx_topic": "uart-bridge/tx" // TX topic
  }
},
"tcp": { // TCP/Websocket settings
  "client": { // Client mode
    "reconnect_interval": 5, // Reconnection interval, seconds
    "remote_addr": "mongoose-os.com:443", // Server HOST:PORT
    "tls": { // TLS settings
      "ca_cert": "ca.pem",
      "cert": "",
      "server_name": "mongoose-os.com"
    },
    "ws": { // Enable websocket mode
      "enable": true,
      "protocol": "",
      "uri": "/uart-to-websocket/device/?????????????" // URI to connect to
    }
  },
  "evict_old": true, // On device ID conflict, evict old entry
  "keepalive": { // Heartbeat settings
    "count": 10,
    "idle": 60,
    "interval": 6
  },
  "listener": { // Server mode settings
    "port": 8910, // Port to listen on
    "tls": { // TLS settings
      "ca_cert": "", // CA certificate for mutual auth
      "cert": "" // Server certificate
    },
    "ws": { // Enable websocket mode
      "enable": false
    }
  },
  "rx_buf_size": 2048, // Receive buffer size, bytes
  "status_interval_ms": 1000, // Status interval, milliseconds
  "tx_buf_size": 2048 // Transmit buffer size, bytes
},
"uart": { // UART settings
  "baud_rate": 115200, // Baud rate
  "rx_buf_size": 1460, // Receive buffer size, bytes
  "rx_fc_ena": false, // Enable flow control
  "rx_linger_micros": 20, // Microseconds to continue reading before sending
  "rx_throttle_when_no_net": true, // Throttle when no network
  "status_interval_ms": 1000, // Status interval, milliseconds
  "swap_rxcts_txrts": false, // Swap RX / CTS and TX / RTS
  "tx_buf_size": 2048, // Transmit buffer size
  "tx_fc_ena": false, // Enable TX flow control
  "uart_no": -1 // UART number, -1 to disable
}

```

In order to change any configuration parameter, execute `mos config-set name=value`, for example:

```

mos config-set mqtt.server=my.server.com:1883
mos config-set kiwi.pub_topic=my_cool_topic

```

Web UI

UART gateway, when not configured, starts its own WiFi Access Point, named `uart-gateway-.....`. The gateway is accessible on IP address 192.168.4.1 in that network.

If WiFi station is configured, then the UART gateway does not start an access point.

In any case, UART gateway runs a Web UI on port 80, allowing to setup tunable parameters over the easy-to-use interface:

- [User guide](intro.md) # Libraries

Mongoose OS libraries are maintained as separate repositories under the [mongoose-os-libs](#) organization.

Here's a list as of 2019/04/30:

Name	Description
adc	ADC support
ads1x1x-i2c	ADS101X and ADS111X I2C drivers
ads7843-spi	
apds9960-i2c	Mongoose OS library for APDS-9960
arduino-adafruit-ads1x15	Arduino Adafruit ADS1015 library for Mongoose OS
arduino-adafruit-bme280	
arduino-adafruit-gfx	
arduino-adafruit-htu21df	
arduino-adafruit-pwm-servo	Arduino Adafruit PWM servo library
arduino-adafruit(ssd1306)	
arduino-adafruit-tsl2561	
arduino-compat	Arduino compatibility library for Mongoose OS
arduino-dallas-temperature	
arduino-ds3231	Arduino DS3231 library for Mongoose OS
arduino-md-parola	
arduino-onewire	
arduino-pololu-vl53l0x	
arduino-spi	
arduino-wire	
atca	
aws	Amazon AWS IoT integration with Mongoose OS
azure	Azure IoT Hub support library for Mongoose OS
barometer	Mongoose OS API and library for barometers
bh1730	BH1730 sensor support
bh1750	
blynk	Blynk integration library for Mongoose OS

bm222	A Bosch BM222 accelerometer library
bme280	
boards	Board configuration defs
bootloader	Boot loader support library
bt-common	Common bits of Bluetooth support
bt-service-config	System Configuration over Bluetooth GATT Service
bt-service-debug	Debug over Bluetooth GATT Service
bt-service-time	Bluetooth current time service implementation
ca-bundle	
ccs811-i2c	Mongoose OS library implementing an I2C driver for CCS811
core	Mongoose OS core lib
cron	
crontab	
dallas-rmt	
dash	
demo-bundle	
dht	
dns-sd	
ds3231	
empty	A blank Mongoose OS library
esp32-touchpad	JS bindings for ESP32 touch pad sensor
ethernet	Ethernet support
file-logger	
freertos	FreeRTOS Kernel
fstab	Storage device and filesystem init table
gcp	Google IoT Core integration for Mongoose OS
http-server	
htu21df-i2c	Mongoose OS library implementing an I2C driver for HTU21D(F)
i2c	
ili9341-spi	ILI9341 TFT Driver for Mongoose OS
imu	Mongoose OS library for inertial measurement units
ina219-i2c	
ina226-i2c	I2C Library for Texas Instruments' current sensor INA226
ina3221-i2c	I2C Library for Texas Instruments' 3-channel current sensor INA3221
ir	IR library for Mongoose OS
js-demo-bundle	

jstore	
location	
lwip	LwIP TCP/IP stack
max7219-spi	MAX7219/MAX7221 SPI based LED display driver.
mbedtls	Optimized version of mbedtls
mcp23xxx	Mongoose OS Library for MCP23x08/MCP23x17: an 8/16-port GPIO expander w/ interrupt support
mcp9808-i2c	Mongoose OS library implementing an I2C driver for MCP9808
md-max72xx	
miniz	Miniz compression library
mjs	Embedded JavaScript engine for Mongoose OS
mongoose	Mongoose Embedded Web Server Library
mq135	MQ135 gas sensor support
mqtt	MQTT protocol support library for Mongoose OS
neopixel	
onewire	
onewire-rmt	
ota-common	OTA common bits
ota-http-client	
ota-http-server	
ota-shadow	
pcf857x-i2c	Mongoose OS Library for PCF8574: an 8-port GPIO expander w/ interrupt support
pppos	PPPoS / cellular modem support
prometheus-metrics	
prometheus-sensors	A library that takes measurements of popular sensors on I2C/SPI and forwards them to Prometheus
provision	A device provisioning library
pwm	
rpc-azure	RPC support for Azure
rpc-common	
rpc-gatts	RPC over GATT, Server
rpc-gcp	RPC support for Google Cloud Platform
rpc-loopback	
rpc-mqtt	Implementation of Mongoose OS RPC over MQTT protocol
rpc-service-atca	
rpc-service-config	
rpc-service-cron	
rpc-service-fs	

rpc-service-gattc	
rpc-service-gpio	
rpc-service-i2c	
rpc-service-ota	
rpc-service-wifi	
rpc-uart	
rpc-ws	WebSocket channel support for mgRPC
sensor-utils	For Mongoose OS - A collection of C-functions with JS wrappers that make it convenient to calculate ...
shadow	
sht31-i2c	Mongoose OS library implementing an I2C driver for SHT31
si7005	Si7005 Temperature and Humidity Sensor Driver for Mongoose OS
si7021-i2c	Mongoose OS library implementing an I2C driver for Si7021
sntp	
spi	Implements SPI API for Mongoose OS
ssd1306	
stlm75	STMicro STLM75 temperature sensor driver
stmpe610-spi	
tmp006	A Texas Instruments TMP006 temperature sensor
veml6075-i2c	
vfs-common	Mongoose OS VFS common bits
vfs-dev-encr	VFS device that encrypts reads and writes
vfs-dev-part	A VFS device that exposes a part of another device
vfs-dev-ram	RAM-based VFS device
vfs-dev-spi-flash	SPI flash VFS device driver
vfs-dev-w25xxx	Winbond W25XXX SPI NAND Flash Driver
vfs-fs-lfs	Mongoose OS support for littlefs
vfs-fs-spiffs	SPIFFS filesystem support
watson	IBM Watson IoT Platform support for Mongoose OS
wifi	
wifi-ism43xxx	Inventek es-WiFi module driver for Mongoose OS
wifi-setup-web-ui	

Heaplog viewer quick overview

So typical use case for the heaplog viewer is when we get some heap integrity violations, something like this in the log:

```
heap integrity broken: block links don't match: 1110 -> 1111, but 1111 -> 0
```

So we enable poisoning, heaplog, and call stack trace, in order to understand better what's going on:

```
$ mos build --build-var MGOS_ENABLE_HEAP_LOG:1 --build-var MGOS_ENABLE_CALL_TRACE:1
```

Before flashing this firmware, make sure you'll be able to save full session's log into a file. I usually adjust MFT `console-line-count` to be really large, say, 50000 lines, and clear the console before flashing the device. It will be easy to just copy-paste the whole log afterwards. Alternatively, you may use `console-log` option of MFT to save the log to the file for you, although you'll have to clean it yourself before each new session.

Then, flash device and reproduce the problem. Now, instead of "heap integrity broken" message, we should get something more concrete about the poison being missing at some exact address, for example:

```
there is no poison after the block. Expected poison address: 0x3fff3dfc,  
actual data: 0x1f 0x00
```

Once reproduced, copy-paste complete session's log from MFT to some file. It should typically begin with something like:

```
--- flashed successfully  
--- connected  
[some garbage omitted]  
rl!◆hlog_param:{ "heap_start":0x3fff01b0, "heap_end":0x3fffc000}
```

The key thing here is `hlog_param:{ "heap_start":0x3fff01b0, "heap_end":0x3fffc000}`

Now, it's very useful to convert all FW addresses to symbolic names. There is a script `heaplog_symbolize.py` for that (located in the same directory as this readme file: `tools/heaplog_viewer`).

Using the heap log server

Heap log server, located under the `heaplog_server` directory, simplifies the process of collecting logs and providing symbols. Once built (you'll need to install [Go](#), you can run it like this:

```
./heaplog_server --logtostderr \  
--document_root .. \  
--binary $(HOME)/cesanta/dev/fw/platforms/esp8266/.build/mongoose-os.out \  
--console_log /tmp/console.log
```

Point it at the firmware binary and console log. Console log can be a serial port or a file that you collected before. After that, you can navigate to <http://localhost:8910/> and use the "Connect to server" button, viewer will load the symbols and log automatically.

If the log file is a file (and not a port), the server tries to automatically find the latest heap log start and uses it as a starting point, so you do not need to clean up the log every time.

Symbolizing manually

Assuming you saved a heaplog to `/tmp/log1`, a command to symbolize it would be:

```
python heaplog_symbolize.py --out-suffix _symb /tmp/log1
```

It will create a new file `/tmp/log1_symb`. If you omit `--out-suffix` option, output will go to stdout.

So now you have a heaplog with symbolic call traces. Open heaplog viewer in your browser: `tools/heaplog_viewer/heaplog_viewer.html` (If you had it already opened with some other data, please refresh it by hitting F5 before loading new file, since viewer is not yet polished)

Click "Choose File", and select the file with log you just saved. After a few seconds, you should see a visual heap map.

NOTE: If the map doesn't appear, your log file might be corrupted, or you might forget to hit F5 before loading new heaplog file. To check if there are some errors, open developers console (in Opera or Chrome, it's Ctrl+Shift+J), and check if there are any uncaught exceptions. If help is needed, ping `dfrank` in Slack.

So now you have a visual heap map and an address where poison is missing (in the example above, it's `0x3fff3dfc`). By moving your mouse over the map, you'll see corresponding addresses. This way it's easy to find the offending address on the map; usually, it's the end of some block. Now we know that this particular block was overflowed. In the details, there will be call stack like the following:

```
#4849 Alloc: 0x3fff70ac, size: 1024, shim: 1, calls: __wrap_pvPortMalloc ←  
malloc ← _malloc_r ← cs_read_file ← init_device
```

Having that information now, it's much easier for you to find a bug. Good luck!

Shortening the log

If the log is too long to be parsed by the heaplog viewer, there is a standalone shortener script which takes a heaplog, parses it, and swallows all the history, leaving just a set of `mallocs` which reconstruct the final picture from the log. Usage:

```
$ cd tools/heaplog_viewer/heaplog_shortener && \  
go build && \  
. ./heaplog_shortener --console_log /path/to/src_log > target_short_log
```

External dependencies for Docker image builds get put here.

Applications

Mongoose OS apps are maintained as separate repositories under the [mongoose-os-apps](#) organization.

Here's a list as of 2019/04/30:

Name	Description
LilyGo-HiGrow-Sensor-v1	LilyGo HiGrow ESP32 Plant Monitoring Sensor Firmware (Hardware v1)
alexa-microwave	Control a microwave with ESP32 via MQTT / Amazon Alexa

aws-iot-button	
aws-iot-heater	
aws-pico	ESP32-PICO-KIT breakout board demo for AWS IoT
aws-uart	
blynk	
blynk-bme280-js	Using BME280 sensor with Blynk mobile app
bootloader	Boot loader
captive-portal	
demo-c	A demo Mongoose OS firmware for C/C++
demo-js	A JS-enabled demo Mongoose OS firmware
door-bell	IoT doorbell that works via MQTT - generic MQTT server, AWS IoT, Google IoT, etc
door-sensor	Mongoose OS IoT Door-Sensor for ESP32
empty	
environmental-monitor	
example-arduino-adafruit-ads1x15-js	Adafruit ADS1015 12-Bit ADC - 4 Channel with Programmable Gain Amplifier usage example (JavaScript)
example-arduino-adafruit-bme280-c	
example-arduino-adafruit-bme280-js	
example-arduino-adafruit-ssd1306-c	
example-arduino-adafruit-ssd1306-js	
example-arduino-dallas-temperature-c	
example-arduino-dallas-temperature-js	
example-arduino-ds3231-js	DS3231 High Precision Real Time Clock Module usage example (JavaScript)
example-arduino-hello-c	
example-arduino-onewire-c	
example-arduino-onewire-js	
example-arduino-wire-c	
example-binary-lib-c	Binary library use example
example-ble-js	BLE JS API use example
example-dht-c	
example-dht-js	
example-dns-sd	DNS-SD usage example
example-m5stack-js	Basic demo for M5stack
example-neopixel-c	
example-neopixel-js	
example-no-libs-c	Bare-bones app with no libraries at all

example-rpc-c	
example-shadow-js	
example-spi-c	C SPI API usage example
example-uart-c	
example-uart-js	
example-wifi-scan-js	WiFi scan app in JS
http-fetch	
huzzah-featherwing	
imu-demo	IMU Library Demo (for Chrome)
lilygo-higrow-soil-temp-humi-sensor	
losant-motion-sensor	Detecting Motion Using a PIR sensor, ESP8266, and Mongoose OS
losant-mqtt	Losant + Mongoose OS example
losant-temp-sensor	A Mongoose OS app for sending temperature readings of ESP devices to Losant.
mq135	
neopixel-aws-iot	Neopixel control using AWS IoT and Android App
ota-aws-shadow	
shelly1	
smallest-wifi-button	The Smallest WiFi Button in the World (ESP8266, MQTT, IFTTT)
smart-pool-pump	Pool Timer Controller, integrates AWS shadow, BMP180 and ADE7912
smart-washing-machine	Making washing machine smart - MQTT, ESP8266
smoke-detector	IoT Smoke Detector Tutorial (IFTTT, SMS)
sonoff-basic-openhab	A Sonoff Basic firmware to work with openHAB
wifi-scan	
wifi-setup-web	

People who have agreed to the

[Cesanta CLA](#)

can make contributions.

We follow the Google C/C++ style guide: <https://google.github.io/styleguide/cppguide.html>

We'd appreciate if your contribution follows the same style guide.

Mongoose OS - an IoT Firmware Development Framework

- Over-The-Air firmware updates and remote management - reliable updates with rollback on failures, remote device access infrastructure
- Security - built in flash encryption, crypto chip support, ARM mbedTLS optimized for small memory footprint
- [Device management dashboard service](#)
- Supported microcontrollers: CC3220, CC3200, ESP32, ESP8266, STM32F4, STM32L4, STM32F7
- Recommended dev kits: [ESP32-DevKitC for AWS IoT](#), [ESP32 Kit for Google IoT Core](#)

- Built-in integration for AWS IoT, Google IoT Core, Microsoft Azure, Adafruit IO, generic MQTT servers
- Code in C or JavaScript
- Ready to go Apps and Libraries
- [Embedded JavaScript engine - mJS](#)

Trusted and Recommended By:

- Amazon AWS - [Amazon AWS Technology Partner](#)
- Google IoT Core - [Mongoose OS is a Google Cloud IoT Core Partner](#)
- IBM Watson IoT - [Mongoose OS is a Ready for IBM Watson IoT validated solution](#)
- Microsoft Azure IoT - [Mongoose OS is recommended by Microsoft Azure IoT](#)
- Texas Instruments - [an official partner of Texas Instruments](#)
- STMicroelectronics - [an official partner of STMicroelectronics](#)
- Espressif Systems - [an official partner of Espressif Systems](#)

Docs, Support

- [Mongoose OS Documentation](#)
- [Support Forum](#) - ask your technical questions here
- [Video tutorials](#)
- [Commercial licensing](#) and [support available](#)

Licensing

Mongoose OS is Open Source and dual-licensed:

- **Mongoose OS Community Edition** - Apache License Version 2.0
- **Mongoose OS Enterprise Edition** - Commercial License

Community vs Enterprise Edition

	Community Edition	Enterprise Edition
License	Apache 2.0	Commercial - contact us
Allows to close end-product's source code	Yes	Yes
Price	Free	Paid, see details
Source code & functionality	Limited	Full
Technical support	Community support via Forum and Chat	Commercial support by Mongoose OS development team, see details

How to contribute

- If you have not done it already, sign [Cesanta CLA](#) and send GitHub pull request.
- Make a Pull Request (PR) against this repo. Please follow [Google Coding Style](#).
Send PR to one of the core team member:
 - [pimvanpelt](#)

- nlviu
- DrBomb
- kzyapkov
- rojer
- cpq

- Responsibilities of the core team members:

- Review and merge PR submissions
- Create new repos in the <https://github.com/mongoose-os-apps> and <https://github.com/mongoose-os-libs> organisations for new app/library contributions
- Create Mongoose OS releases

Flashing ESP8266

Wiring

At the bare minimum, you need to have serial TX & RX connected to the RX and TX pins on ESP8266 board. To enter flashing mode GPIO0 must be pulled down on boot, so you can simply connect it to GND before connecting power.

Other option is to connect GPIO0 to DTR (Data Terminal Ready) line of serial adapter and RESET pin to RTS (Request To Send) line. This is the way it is wired in NodeMCU boards and what `esptool.py` also assumes.

Summary:

- GND - ground
- TX - serial RX
- RX - serial TX
- VCC - power
- CH_PD - power
- GPIO0 - ground or serial DTR
- RESET - leave unconnected or serial RTS

Drivers

Don't forget to install proper drivers for the USB-to-serial converter.

If you're using a board along with a standalone FTDI adapter on Linux or Mac OS X, it should be shipped with the OS, for Windows drivers see [FTDI website](#).

NodeMCU v2 boards come with Silabs CP2102 USB-to-serial chip for which you can get driver from [here](#).

NodeMCU 0.9 and some other boards use the CH34x chip, Win7 driver can be found [here](#).

Mongoose Flashing Tool

If you have DTR and RTS connected properly, pressing "Load firmware" button should reboot the device automatically. Otherwise, just connect GPIO0 to ground and reset the device manually before flashing.

esptool

As `esptool.py` does not generate the device ID, you need to do this manually if you want to connect the device to our cloud. To do this invoke `mkid.py` script:

```
./tools/mkid.py --id ${ID} --psk ${PSK} > 0x10000.bin
```

Replace `${ID}` with a few random characters and `${PSK}` with a password unique to this device. Alternatively, you can generate a random ID with MFT:

```
MFT --generate-id 0x10000.bin
```

After that just flash `0x10000.bin` at offset `0x10000` along with other pieces:

```
path/to/esptool.py -b 115200 -p /dev/ttyUSB0 write_flash 0x00000 0x00000.bin 0x10000 0x10000.bin 0x1d000 0x1d000.bin 0x6d000 0x6d000.bin
```

For NodeMCU v2 board don't forget to also add

`--flash_mode dio --flash_size 32m` flags after `write_flash` or the device won't boot.

```
/*
```

- Copyright (c) 2016, Texas Instruments Incorporated
- All rights reserved.
-
- Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
 - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
 - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
 -
 - Neither the name of Texas Instruments Incorporated nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
 -
 - THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 - THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 - /
 - /
- ===== CC3220SF_LAUNCHXL.cmd =====
- */

```
--stack_size=1024  
--heap_size=0 /* minimize heap since we are using heap_4.c */
```

```
--entry_point=_c_int00
```

```
/*
```

- The starting address of the application. Normally the interrupt vectors
 - must be located at the beginning of the application.
- ```
 */
```

## define SRAM\_BASE 0x20000000

## define FLASH\_BASE 0x01000800

```
MEMORY
```

```
{
```

```
/* Bootloader uses FLASH_HDR during initialization /
```

```
FLASH_HDR (RX) : origin = 0x01000000, length = 0x7FF / 2 KB /
```

```
FLASH (RX) : origin = 0x01000800, length = 0x0FF800 / 1022KB /
```

```
SRAM (RWX) : origin = 0x20000000, length = 0x00040000 / 256KB */
```

```
}
```

```
/* Section allocation in memory */
```

```
SECTIONS
```

```
{
```

```
.dbghdr : > FLASH_HDR
```

```
.binit : > FLASH
```

```
.text : > FLASH
```

```
.text.* : > FLASH
```

```
.const : > FLASH
```

```
.cinit : > FLASH
```

```
.pinit : > FLASH
```

```
.init_array : > FLASH
```

```
GROUP {
 .int_vecs
 .TI.ramfunc .iram .iram.*
} load=FLASH, run=SRAM_BASE, table(BINIT)
GROUP {
 .bss
 .data
 .heap_start
} > SRAM
GROUP {
 .heap_end
 .stack
} > SRAM(HIGH)
```

```
}
```

## Mongoose OS Firmware boot loader for CC3200

`mg-boot.bin` should be loaded at `0x20004000` (this is where ROM loads `/sys/mcuimg.bin`).

It consists of a small relocator followed by the loader body.

The relocator, which is executed first, moves the loader body

to 0x2003C000 (256K - 16K) and executes it from there.

The loader's stack starts at 0x2003C000 and grows down.

Loader then proceeds to initialize NWP and read its configuration.

Configuration, in the form of `struct boot_cfg`, is read either from `mg-boot.cfg.0` or `.1`. If both exist, whichever is more recent is used.

Recency is determined by comparing the `seq` counter. NB: config with *smaller* `seq` is deemed to be more recent (counts down from `BOOT_CFG_INITIAL_SEQ`).

The config specifies which image to load next (`image_file`) and where to (`base_address`). As usual, it is assumed that the image starts with interrupt stack pointer followed by interrupt vector table.

Relocating loader to the end of RAM allows using the top 16 KB so application images starting at 0x20000000 are perfectly fine.

Images up to approximately 240KB (256KB - 16KB - loader stack) can be loaded this way.

Boot configs can be created with the `mkcfg` tool located in the `tools` dir:

```
$ mkcfg $IMAGE_FILE $BASE_ADDRESS > boot.cfg.0
```

This initializes `seq` to `BOOT_CFG_INITIAL_SEQ`. If specifying `seq` is required:

```
$ mkcfg $IMAGE_FILE $BASE_ADDRESS $SEQ > boot.cfg.0
```

## CC3200 ROM communication protocol

All communication is done over one serial port at baud rate 921600 (no autodetection is performed). In most situations data is transmitted in frames, each frame must be acknowledged by the remote side (by sending 2 bytes: "\x00\xCC").

### Frame format

Frames are length-prefixed, with no delimiters inbetween.

| Length  | Description                                                    |
|---------|----------------------------------------------------------------|
| 2 bytes | big-endian number, length of payload plus length of this field |
| 1 byte  | checksum, sum of all payload bytes modulo 256                  |
| N bytes | payload                                                        |

First byte of the payload usually is an opcode, rest is the arguments (numbers are big-endian, if not stated explicitly).

Before sending any frames to a freshly booted board you need to send break first, then read back one ACK ("\x00\xCC").

### Known opcodes

Opcodes are sorted by numerical value.

- 0x21 – start file upload.
  - Arguments:
    - 2 bytes: 0
    - 1 byte: 0x30 + block size (0 – 256, 1 – 1024, 2 – 4096, 3 – 16384)
    - 1 byte: number of blocks (in terms of the above block size)
    - N bytes: target filename (usually /sys/mcuimg.bin) followed by 2 zero bytes
  - After sending ACK the device will send 4 zero bytes.
- 0x22 – finish file upload.
  - Arguments:
    - 63 bytes: all 0
    - 256 bytes: all 0x46
    - 1 byte: 0
- 0x24 – file chunk. Each chunk carries 4096 bytes of the file (not the block size specified in 0x21).
  - Arguments:
    - 4 bytes: offset of the chunk
    - 4096 bytes: data
- 0x2A – get file info.
  - Arguments:
    - 4 bytes: name length
    - N bytes: file name
  - Response:
    - 24 bytes: first byte is 1 if the file exists, 4 bytes at offset 4 – file size
- 0x2B – get file chunk.
  - Arguments:
    - 4 bytes: offset
    - 4 bytes: number of bytes to read
  - Response:
    - N bytes: content
- 0x2D – raw storage write.
  - Arguments:
    - 4 bytes: storage ID – 0
    - 4 bytes: offset
    - 4 bytes: length
    - N bytes: data
- 0x2E – file deletion.
  - Arguments:
    - 4 bytes: 0
    - N bytes: zero-terminated filename.
- 0x2F – version info.
  - Response is sent in a new frame that you must ACK:
    - 28 bytes: some structure with first 4 bytes containing bootloader version.
- 0x30 – raw storage block erasing.
  - Arguments:
    - 4 bytes: storage ID – 0
    - 4 bytes: index of the first block to erase (block size can be determined with command 0x31)
    - 4 bytes: number of blocks to erase
- 0x31 – get storage info.
  - Arguments:
    - 4 bytes: storage ID – 0
  - Response is sent in a new frame that you must ACK:
    - 2 bytes: block size
    - 2 bytes: number of blocks
- 0x32 – exec. Sometimes you need to read 2 ACKs instead of one.

- 0x33 – switch UART pins to another port.
  - Arguments:
    - 4 bytes: magic number 0x0196E6AB

/\*

- Copyright (c) 2014-2016 Cesanta Software Limited
- All rights reserved
- 
- Linker command file for the TI compiler.

\*/

--retain=g\_pfnVectors

/\* External vars: APP\_ORG SRAM\_BASE\_ADDR SRAM\_SIZE \*/

/\* System memory map \*/

## MEMORY

```
{
SRAM (RWX) : origin = APP_ORG, length = (SRAM_SIZE - (APP_ORG - SRAM_BASE_ADDR))
}
```

/\* Section allocation in memory \*/

## SECTIONS

```
{
.intvecs: > APP_ORG
GROUP {
.init_array
.vtable
.iram .iram.*
.text .text.*
.const
.data
.bss_start
.bss
.bss_end
.pinit
.heap_start
} > SRAM
GROUP {
.sysmem : type = DSECT
.heap_end
.stack
} > SRAM(HIGH)
}
```

# Simple device control using MQTT

This example demonstrates how to perform remote device control over MQTT protocol. Build & flash a firmware and configure WiFi (see [quick start guide](#)), and configure MQTT server:

```
mos build --arch esp8266
mos flash
mos config-set wifi.....
mos console
```

Then use an MQTT client to send messages, for example web-based client at

<http://www.hivemq.com/demos/websocket-client/>. Click on "Connect", then

"Add new topic subscription", choose topic /response, then publish a message. In the "Topic" field, specify /request, and in the "Message" field, one of the following:

- {"gpio": {"pin": 2, "state": 0}} - sets GPIO pin X into state Y
- {"button": {"pin": 0}} - start listening on button presses on GPIO pin X. When button is pressed, a message "click" is sent to the /response topic
- {"i2c\_write": {"data": "af404c"}} - write byte stream to the I2C bus.  
In this example, there are 3 bytes 0xaf 0x40 0x4c.  
First byte is an I2C address. Status code 0 in response means success, all other codes mean failure.

By default, this firmware connects to the broker.mqttdashboard.com:1883 MQTT server.

You can change this default setting by editing src/conf\_schema.yaml file and rebuilding/reflashing the firmware. Alternatively, you can change any setting on the fly, for example to choose a different MQTT server:

```
mos config-set mqtt.server=my_server:my_port
```

## How to use Amazon IoT with this example

First, download the Amazon's aws utility and run aws configure.

Then you're ready to onboard

```
... build, flash, configure wifi
mos aws-iot-setup --aws-iot-policy YOUR_POLICY
mos console
```

Login to AWS IoT console, use web-based MQTT client to send messages.

See <https://mongoose-os.com/blog/esp8266-remote-control-via-aws-iot/> for the step-by-step instructions.



## JSON parser and emitter for C/C++

## Features

- ISO C and ISO C++ compliant portable code
- Very small footprint
- No dependencies
- json\_scanf() scans a string directly into C/C++ variables
- json\_printf() prints C/C++ variables directly into an output stream

- `json_setf()` modifies an existing JSON string
- `json_fread()` reads JSON from a file
- `json_fprintf()` writes JSON to a file
- Built-in base64 encoder and decoder for binary data
- Parser provides low-level callback API and high-level `scanf`-like API
- 100% test coverage
- Used in [Mongoose OS](#), an operating system  
for connected commercial products on low-power microcontrollers

# API reference

## `json_scanf()`, `json_vscanf`

```
int json_scanf(const char *str, int str_len, const char *fmt, ...);
int json_vscanf(const char *str, int str_len, const char *fmt, va_list ap);

/* json_scanf's %M handler */
typedef void (*json_scanner_t)(const char *str, int len, void *user_data);
```

Scans the JSON string `str`, performing `scanf`-like conversions according to `fmt`.

`fmt` uses `scanf()`-like format, with the following differences:

1. Object keys in the format string don't have to be quoted, e.g. "{key: %d}"
2. Order of keys in the format string does not matter, and the format string may omit keys to fetch only those that are of interest, for example,  
assume `str` is a JSON string `{ "a": 123, "b": "hi", c: true }`.

We can fetch only the value of the `c` key:

```
int value = 0;
json_scanf(str, strlen(str), "{c: %B}", &value);
```

3. Several extra format specifiers are supported:

- `%B`: consumes `int *` (or `char *`, if `sizeof(bool) == sizeof(char)`), expects boolean `true` or `false`.
- `%Q`: consumes `char **`, expects quoted, JSON-encoded string. Scanned string is malloc-ed, caller must free() the string.
- `%V`: consumes `char **`, `int *`. Expects base64-encoded string.  
Result string is base64-decoded, malloced and NUL-terminated.  
The length of result string is stored in `int *` placeholder.  
Caller must free() the result.
- `%H`: consumes `int *, char **`.  
Expects a hex-encoded string, e.g. "fa014f".  
Result string is hex-decoded, malloced and NUL-terminated.  
The length of the result string is stored in `int *` placeholder.  
Caller must free() the result.
- `%M`: consumes custom scanning function pointer and  
`void *user_data` parameter - see `json_scanner_t` definition.
- `%T`: consumes `struct json_token *`, fills it out with matched token.

Returns the number of elements successfully scanned & converted.

Negative number means scan error.

Example - scan arbitrary JSON string:

```

// str has the following JSON string (notice keys are out of order):
// { "a": 123, "d": true, "b": [1, 2], "c": "hi" }

int a = 0, d = 0;
char *c = NULL;
void *my_data = NULL;
json_scanf(str, strlen(str), "{ a:%d, b:%M, c:%Q, d:%B }",
 &a, scan_array, my_data, &c, &d);

// This function is called by json_scanf() call above.
// str is "[1, 2]", user_data is my_data.
static void scan_array(const char *str, int len, void *user_data) {
 struct json_token t;
 int i;
 printf("Parsing array: %.*s\n", len, str);
 for (i = 0; json_scanf_array_elem(str, len, "", i, &t) > 0; i++) {
 printf("Index %d, token [%.*s]\n", i, t.len, t.ptr);
 }
}

```

Example - parse array of objects:

```

// str has the following JSON string - array of objects:
// { "a": [{"b": 123}, {"b": 345}] }
// This example shows how to iterate over array, and parse each object.

int i, value, len = strlen(str);
struct json_token t;

for (i = 0; json_scanf_array_elem(str, len, ".a", i, &t) > 0; i++) {
 // t.type == JSON_TYPE_OBJECT
 json_scanf(t.ptr, t.len, "{b: %d}", &value); // value is 123, then 345
}

```

## json\_scanf\_array\_elem()

```

int json_scanf_array_elem(const char *s, int len,
 const char *path,
 int index,
 struct json_token *token);

```

A helper function to scan an array item with given path and index.

Fills `token` with the matched JSON token.

Returns 0 if no array element found, otherwise non-0.

## json\_printf()

Frozen printing API is pluggable. Out of the box, Frozen provides a way to print to a string buffer or to an opened file stream. It is easy to tell Frozen to print to another destination, for example, to a socket, etc. Frozen does this by defining an "output context" descriptor which has a pointer to a low-level printing function. If you want to print to another destination, just define your specific printing function and initialise output context with it.

This is the definition of the output context descriptor:

```

struct json_out {
 int (*printer)(struct json_out *, const char *str, size_t len);
 union {
 struct {
 char *buf;
 size_t size;
 size_t len;
 } buf;
 void *data;
 FILE *fp;
 } u;
};

```

Frozen provides two helper macros to initialise two built-in output descriptors:

```

struct json_out out1 = JSON_OUT_BUF(buf, len);
struct json_out out2 = JSON_OUT_FILE(fp);

```

```

typedef int (*json_printf_callback_t)(struct json_out *, va_list *ap);
int json_printf(struct json_out *, const char *fmt, ...);
int json_vprintf(struct json_out *, const char *fmt, va_list ap);

```

Generate formatted output into a given string buffer, auto-escaping keys.

This is a superset of printf() function, with extra format specifiers:

- %B print json boolean, true or false. Accepts an int.
- %Q print quoted escaped string or null. Accepts a const char \*.
- %.\*Q same as %Q, but with length. Accepts int, const char \*
- %V print quoted base64-encoded string. Accepts a const char \*, int.
- %H print quoted hex-encoded string. Accepts a int, const char \*.
- %M invokes a json\_printf\_callback\_t function. That callback function can consume more parameters.

Return number of bytes printed. If the return value is bigger than the supplied buffer, that is an indicator of overflow. In the overflow case, overflow bytes are not printed.

Example:

```

json_printf(&out, "{%Q: %d, x: [%B, %B], y: %Q}", "foo", 123, 0, -1, "hi");
// Result:
// {"foo": 123, "x": [false, true], "y": "hi"}

```

To print a complex object (for example, serialise a structure into an object), use %M format specifier:

```

struct my_struct { int a, b; } mys = {1,2};
json_printf(&out, "{foo: %M, bar: %d}", print_my_struct, &mys, 3);
// Result:
// {"foo": {"a": 1, "b": 2}, "bar": 3}

```

```

int print_my_struct(struct json_out *out, va_list *ap) {
 struct my_struct *p = va_arg(*ap, struct my_struct *);
 return json_printf(out, "{a: %d, b: %d}", p->a, p->b);
}

```

## json\_printf\_array()

```
int json_printf_array(struct json_out *, va_list *ap);
```

A helper %M callback that prints contiguous C arrays.

Consumes void \*array\_ptr, size\_t array\_size, size\_t elem\_size, char \*fmt

Returns number of bytes printed.

## json\_walk() - low level parsing API

```
/* JSON token type */
enum json_token_type {
 JSON_TYPE_INVALID = 0, /* memsetting to 0 should create INVALID value */
 JSON_TYPE_STRING,
 JSON_TYPE_NUMBER,
 JSON_TYPE_TRUE,
 JSON_TYPE_FALSE,
 JSON_TYPE_NULL,
 JSON_TYPE_OBJECT_START,
 JSON_TYPE_OBJECT_END,
 JSON_TYPE_ARRAY_START,
 JSON_TYPE_ARRAY_END,
 JSON_TYPES_CNT,
};

/*
 * Structure containing token type and value. Used in `json_walk()` and
 * `json_scanf()` with the format specifier `%T` .
 */
struct json_token {
 const char *ptr; /* Points to the beginning of the value */
 int len; /* Value length */
 enum json_token_type type; /* Type of the token, possible values are above */
};

/* Callback-based API */
typedef void (*json_walk_callback_t)(void *callback_data,
 const char *name, size_t name_len,
 const char *path,
 const struct json_token *token);

/*
 * Parse `json_string` , invoking `callback` in a way similar to SAX parsers;
 * see `json_walk_callback_t` .
 */
int json_walk(const char *json_string, int json_string_length,
 json_walk_callback_t callback, void *callback_data);
```

json\_walk() is a low-level, callback based parsing API.

json\_walk() calls a given callback function for each scanned value.

Callback receives a name, a path to the value, a JSON token that points to the value and an arbitrary user data pointer.

The path is constructed using this rule:

- Root element has "" (empty string) path
- When an object starts, . (dot) is appended to the path
- When an object key is parsed, a key name is appended to the path
- When an array is parsed, an [ELEMENT\_INDEX] is appended for each element

For example, consider the following json string:

```
{ "foo": 123, "bar": [1, 2, { "baz": true }] }.
```

The sequence of callback invocations will be as follows:

- type: JSON\_TYPE\_OBJECT\_START, name: NULL, path: "", value: NULL
- type: JSON\_TYPE\_NUMBER, name: "foo", path: ".foo", value: "123"
- type: JSON\_TYPE\_ARRAY\_START, name: "bar", path: ".bar", value: NULL
- type: JSON\_TYPE\_NUMBER, name: "0", path: ".bar[0]", value: "1"
- type: JSON\_TYPE\_NUMBER, name: "1", path: ".bar[1]", value: "2"
- type: JSON\_TYPE\_OBJECT\_START, name: "2", path: ".bar[2]", value: NULL
- type: JSON\_TYPE\_TRUE, name: "baz", path: ".bar[2].baz", value: "true"
- type: JSON\_TYPE\_OBJECT\_END, name: NULL, path: ".bar[2]", value: "{ \"baz\": true }"
- type: JSON\_TYPE\_ARRAY\_END, name: NULL, path: ".bar", value: "[ 1, 2, { \"baz\": true } ]"
- type: JSON\_TYPE\_OBJECT\_END, name: NULL, path: "", value: "{ \"foo\": 123, \"bar\": [ 1, 2, { \"baz\": true } ] }"

If top-level element is an array: [1, {"foo": 2}]

- type: JSON\_TYPE\_ARRAY\_START, name: NULL, path: "", value: NULL
- type: JSON\_TYPE\_NUMBER, name: "0", path: "[0]", value: "1"
- type: JSON\_TYPE\_OBJECT\_START, name: "1", path: "[1]", value: NULL
- type: JSON\_TYPE\_NUMBER, name: "foo", path: "[1].foo", value: "2"
- type: JSON\_TYPE\_OBJECT\_END, name: NULL, path: "[1]", value: "{\"foo\": 2}"
- type: JSON\_TYPE\_ARRAY\_END, name: NULL, path: "", value: "[1, {\"foo\": 2}]"

If top-level element is a scalar: true

- type: JSON\_TYPE\_TRUE, name: NULL, path: "", value: "true"

## json\_fprintf(), json\_vfprintf()

```
/*
 * Same as json_printf, but prints to a file.
 * File is created if does not exist. File is truncated if already exists.
 */
int json_fprintf(const char *file_name, const char *fmt, ...);
int json_vfprintf(const char *file_name, const char *fmt, va_list ap);
```

## json\_asprintf(), json\_vasprintf()

```
/*
 * Print JSON into an allocated 0-terminated string.
 * Return allocated string, or NULL on error.
 * Example:
 *
 * ````c
 * char *str = json_asprintf("{a:%H}", 3, "abc");
 * printf("%s\n", str); // Prints "616263"
 * free(str);
 * ````
 */
char *json_asprintf(const char *fmt, ...);
char *json_vasprintf(const char *fmt, va_list ap);
```

## json\_fread()

```
/*
 * Read the whole file in memory.
 * Return malloc-ed file content, or NULL on error. The caller must free().
 */
char *json_fread(const char *file_name);
```

## json\_setf(), json\_vsetf()

```
/*
 * Update given JSON string `s,len` by changing the value at given `json_path` .
 * The result is saved to `out` . If `json_fmt` == NULL, that deletes the key.
 * If path is not present, missing keys are added. Array path without an
 * index pushes a value to the end of an array.
 * Return 1 if the string was changed, 0 otherwise.
 *
 * Example: s is a JSON string { "a": 1, "b": [2] }
 * json_setf(s, len, out, ".a", "7"); // { "a": 7, "b": [2] }
 * json_setf(s, len, out, ".b", "7"); // { "a": 1, "b": 7 }
 * json_setf(s, len, out, ".b[]", "7"); // { "a": 1, "b": [2,7] }
 * json_setf(s, len, out, ".b", NULL); // { "a": 1 }
 */
int json_setf(const char *s, int len, struct json_out *out,
 const char *json_path, const char *json_fmt, ...);

int json_vsetf(const char *s, int len, struct json_out *out,
 const char *json_path, const char *json_fmt, va_list ap);
```

## json\_prettyify()

```
/*
 * Pretty-print JSON string `s,len` into `out` .
 * Return number of processed bytes in `s` .
 */
int json_prettyify(const char *s, int len, struct json_out *out);
```

## json\_prettyify\_file()

```
/*
 * Prettify JSON file `file_name` .
 * Return number of processed bytes, or negative number of error.
 * On error, file content is not modified.
 */
int json_prettyify_file(const char *file_name);
```

## json\_next\_key(), json\_next\_elem()

```

/*
 * Iterate over an object at given JSON `path`.
 * On each iteration, fill the `key` and `val` tokens. It is OK to pass NULL
 * for `key` , or `val` , in which case they won't be populated.
 * Return an opaque value suitable for the next iteration, or NULL when done.
 *
 * Example:
 *
 * ````c
 * void *h = NULL;
 * struct json_token key, val;
 * while ((h = json_next_key(s, len, h, ".foo", &key, &val)) != NULL) {
 * printf("[%.*s] -> [%.*s]\n", key.len, key.ptr, val.len, val.ptr);
 * }
 * ```
 */
void *json_next_key(const char *s, int len, void *handle, const char *path,
 struct json_token *key, struct json_token *val);

/*
 * Iterate over an array at given JSON `path` .
 * Similar to `json_next_key` , but fills array index `idx` instead of `key` .
 */
void *json_next_elem(const char *s, int len, void *handle, const char *path,
 int *idx, struct json_token *val);

```

## Minimal mode

By building with `-DJSON_MINIMAL=1` footprint can be significantly reduced.

The following limitations apply in this configuration:

- Only integer numbers are supported. This affects parsing and `%f/%lf` conversions in `printf` and `scanf`.
- Hex ('%H') and base64 (%V) conversions are disabled.

## Examples

### Print JSON configuration to a file

```

json_fprintf("settings.json", "{ a: %d, b: %Q }", 123, "string_value");
json_prettyfile("settings.json"); // Optional

```

### Read JSON configuration from a file

```

struct my_config { int a; char *b; } c = { .a = 0, .b = NULL };
char *content = json_fread("settings.json");
json_scanf(content, strlen(content), "{a: %d, b: %Q}", &c.a, &c.b);

```

### Modify configuration setting in a JSON file

```
const char *settings_file_name = "settings.json", *tmp_file_name = "tmp.json";
char *content = json_fread(settings_file_name);
FILE *fp = fopen(tmp_file_name, "w");
struct json_out out = JSON_OUT_FILE(fp);
json_setf(content, strlen(content), &out, ".b", "%Q", "new_string_value");
fclose(fp);
json_prettyify_file(tmp_file_name); // Optional
rename(tmp_file_name, settings_file_name);
```

# Contributions

To submit contributions, sign

[Cesanta CLA](#)

and send GitHub pull request.

# Licensing

Frozen is released under the

[Apache 2.0 license](#).

For commercial support and professional services,

[contact us](#).