

# Cloud Communication Module (CCM)



**User Guide**

**Copyright (C) Cesanta 2018**

# Table of Contents

CCM-MOD module .....	1
Overview .....	1
Wiring .....	1
Communication protocol .....	1
mjson JSON-RPC library .....	2
RPC Services provided by the CCM .....	3
Shadow.Report .....	3
MQTT.Pub .....	4
RPC Services provided by the Host .....	4
RPC.List .....	4
Sys.Info .....	4
Shadow.Delta .....	4
Provisioning CCM module .....	5
WiFi provisioning .....	5
WiFi provisioning via Access Point .....	5
WiFi provisioning via Serial line .....	5
Cloud provisioning .....	5
AWS IoT provisioning .....	5
Microsoft Azure provisioning .....	6
Google IoT Core provisioning .....	6
IBM Watson provisioning .....	6
Factory reset .....	6
Security .....	6
Configuration reference .....	6
CCM-EVAL evaluation board .....	8
Wiring .....	8
Load Arduino sketch .....	8

# CCM-MOD module

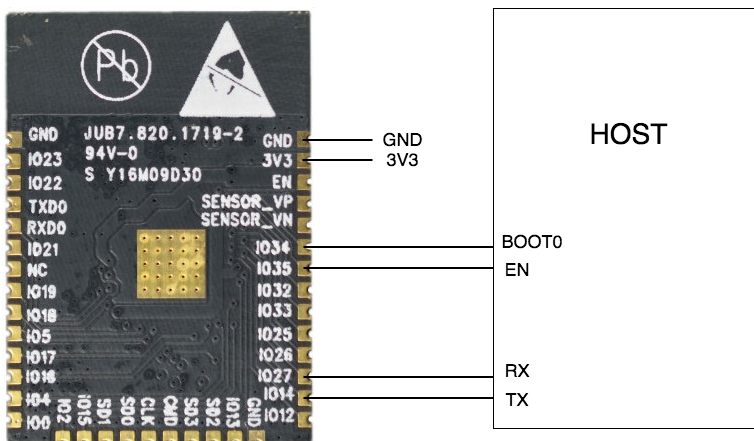
## Overview

CCM module provides secure IoT connectivity for any microcontroller-based product. Below is the summary of the provided functionality:

- Send MQTT messages to the cloud
- Subscribe to MQTT messages from the cloud
- Update device shadow / device twin (AWS, Azure, mDash)
- Receive device shadow / device twin updates (AWS, Azure, mDash)
- Provision CCM over the Bluetooth Low Energy (BLE)
- Provision CCM over the WiFi Access Point
- Provision CCM from the Host MCU
- Update Host MCU over-the-air. Implemented for STM32 or AVR Hosts
- Update CCM itself over-the-air

## Wiring

Host microcontroller must be connected to the CCM via 4 lines. The TX/RX lines provide UART communication between Host and CCM. The BOOT and CN lines enable CCM to move Host into the bootloader mode to update Host's firmware. The diagram below shows the bottom view of the CCM module. CCM pins 27 and 14 must be wired to the TX, RX pins of the Host, respectively. CCM pins 34, 35 must be wired to BOOTo, EN pins respectively:



## Communication protocol

Host MCU communicates with CCM over the UART. The communication protocol is JSON-RPCv2, where JSON-RPC frames are delimited by newline characters. Per JSON-RPC standard, JSON frames with the `id` attribute require response, whereas frames without the `id` attribute are notifications that do not trigger any response.

For example, CCM sends a device shadow delta to the Host. This frame does not have an `id` attribute, thus Host does not send a response:

```
{"method": "Shadow.Delta", "params": {"on": true}} // CCM -> Host
```

When Host receives a JSON-RPC frame, it must parse the frame and call corresponding handler function. That is done by the

In the following example, a CCM calls a custom function `Sum` that adds two numbers:

```
{"id": 12007, "method": "Sum", "params": [2,3]} // CCM -> Host
{"id": 12007, "result": 5} // Host -> CCM
```

And here is the example when CCM calls a non-existent function:

```
{"id": 12008, "method": "This_Function_Does_Not_Exist", "params": true}
{"id": 12008, "error": {"code": -32601, "message": "method not found"}}
```

The communication is two-way. This is an example where Host sends an MQTT message to the cloud:

```
{"method": "MQTT.Pub", "params": {"topic": "t1", "message": "hello"}}
```

## mjson JSON-RPC library

For the Host, a compact C/C++ JSON-RPC library called `mjson` is available. `mjson` is a single-header, open source, MIT-licensed library located at <https://github.com/cesanta/mjson>. `mjson` makes it easy to define RPC services on the Host MCU, and call RPC services on CCM.

The gist of the `msjson` usage is this:

- Initialise the library: call `jsonrpc_init()`, then export any number of the custom functions using `jsonrpc_export()`.
- In the event loop, read the UART in a buffer. When a newline is received, call `json_process()`.
- At any time, you can call `jsonrpc_call()` to send a JSON-RPC frame to the CCM.

The Arduino sketch below demonstrates how to create an RPC service called `Sum` on the Host:

```

#include "mjson.h" // Sketch -> Add file -> add mjson.h

// Gets called by the RPC engine to send a reply frame
static int sender(const char *frame, int frame_len, void *privdata) {
    return Serial.write(frame, frame_len);
}

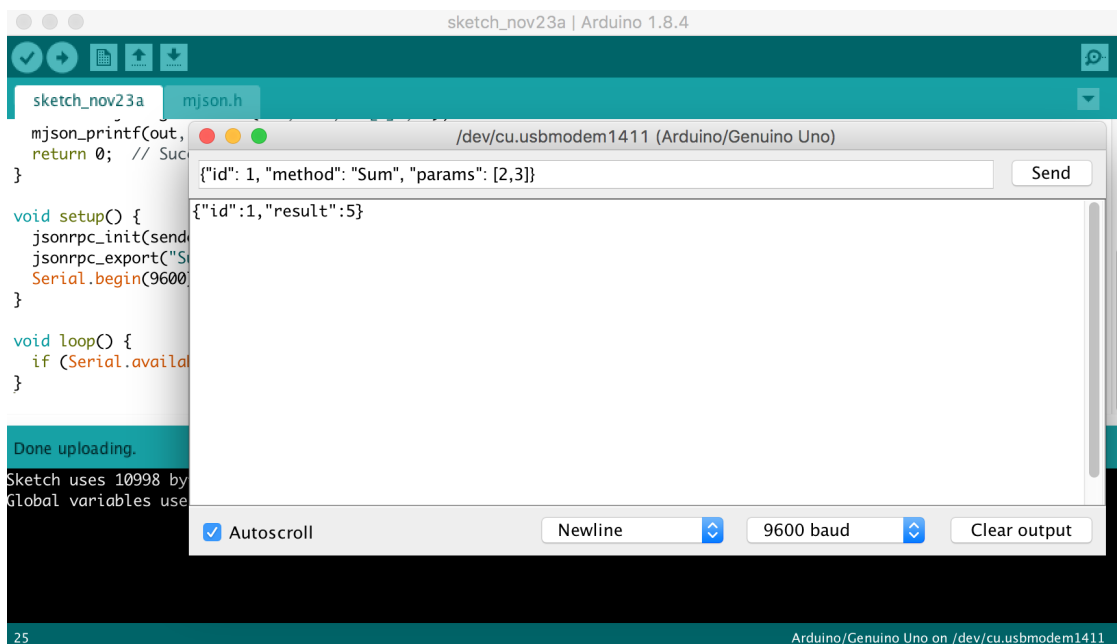
// RPC handler for "Sum". Expect an array of two integers in "params"
static void sum(struct jsonrpc_request *r) {
    int a = mjson_get_number(r->params, r->params_len, "$[0]", 0);
    int b = mjson_get_number(r->params, r->params_len, "$[1]", 0);
    jsonrpc_return_success(r, "%d", a + b);
}

void setup() {
    jsonrpc_init(sender, NULL, NULL, "1.0"); // Initialise the library
    jsonrpc_export("Sum", sum, NULL);      // Export "Sum" function
    Serial.begin(115200);                  // Setup serial port
}

void loop() {
    if (Serial.available() > 0) jsonrpc_process_byte(Serial.read());
}

```

When this sketch is compiled and flashed on an Arduino board, start Arduino Serial Monitor, type `{"id": 1, "method": "Sum", "params": [2,3]}` and hit enter. You should see an answer frame:



## RPC Services provided by the CCM

### Shadow.Report

Update device's `reported` state on the cloud. Parameters: arbitrary JSON value - a string, number, array, or an object of an arbitrary complexity.

Listing 1. Update shadow reported state on the cloud: set "reported.on" to true

```

jsonrpc_call("{\"method\": \"Shadow.Report\", \"params\": {\"on\": true}}", "on", "true");

```

Listing 2. Corresponing JSON-RPC frame generated by the above call (note the absence of the `id`):

```
{"method": "Shadow.Report", "params": {"on": true}}
```

## MQTT.Pub

Send MQTT message to the cloud. Parameters: a JSON object `{"topic": xx, "message": xx, "qos": xx}`. `topic` and `message` must be strings, and `qos` must be a number 0 or 1.

Listing 3. Update shadow reported state on the cloud: set "reported.on" to true

```
jsonrpc_call("{\"method\":\"MQTT.Pub\",\"params\":{\"topic\":\"t1\",\"message\":\"hello\",\"qos\":1}\",  
            \"topic\", \"t1\", \"message\", \"hello\", \"qos\", 1);
```

Listing 4. Corresponing JSON-RPC frame generated by the above call (note the absence of the `id`):

```
{"method": "MQTT.Pub", "params": {"topic": "t1", "message": "hello", "qos": 1}}
```

## RPC Services provided by the Host

### RPC.List

Return a list of all RPC services exported by the Host. Parameters: none.



This is a built-in service.

Serial communication example:

```
{"id": 1, "method": "RPC.List"} // CCM -> Host  
{"id":1,"result":["Shadow.Delta","RPC.List","Sys.Info"]} // Host -> CCM
```

### Sys.Info

Return information about the Host. Parameters: none.



This is a built-in service.

Serial communication example:

```
{"id":1,"method": "Sys.Info"} // CCM -> Host  
{"id":1,"result":{"fw_version":"1.0", "arch":"unknown", "fw_id":"Nov 24 2018 09:10:27", "app"  
:"posix_device"}} // Host -> CCM
```

### Shadow.Delta

React on the device shadow change. This is a notification service, meaning that the handler function must not send any return value. Instead, it could send a `Shadow.Report` notification to the cloud, informing the cloud about the changed state.



This service should be implemented by the user if user wants the device to be shadow-controlled.

Parameters: arbitrary JSON value - a string, number, array, or an object of an arbitrary complexity.

Example serial communication (note the absence of the `id`):

```
{"method": "Shadow.Delta", "params": {"on": true}} // CCM -> Host
{"method": "Shadow.Report", "params": {"on": true, "uptime": 132}} // Host -> CCM
```

## Provisioning CCM module

### WiFi provisioning

#### WiFi provisioning via Access Point

When an unprovisioned CCM module boots up, it can open a WiFi Access Point. It is possible to connect to the Access Point and use a RESTful service to setup WiFi. An un-provisioned CCM is accessible at predefined IP address `192.168.4.1`. Example WiFi setup:

1. Connect to a WiFi network `CCM-xxxxxx`
2. Run the following command (replace NAME and PASS with your WiFi name and password):

```
curl -d '{"config": {"wifi": {"sta": {"enable": true, "ssid": "NAME", "pass": "PASS"}}}}'
http://192.168.4.1/rpc
```

#### WiFi provisioning via Serial line

Using this method, a CCM module could be provisioned by the Host controller. Call two RPC methods: `Config.Set` followed by the `Config.Save`:

Listing 5. Update CCM configuration - setup local WiFi network

```
jsonrpc_call("{\"method\":\"Config.Set\",\"params\":{\"config\":{\"wifi\":{\"sta\":{\"enable\":true,\"ssid\":\"%Q\",\"pass\":\"%Q\"}}}}\",
              \"NAME\", \"PASS\");
jsonrpc_call("{\"method\":\"Config.Save\",\"params\":{\"reboot\":true});
```

Listing 6. Corresponding JSON-RPC frame generated by the above call (note the absence of the `id`):

```
{"id":1,"method":"Config.Set","params":{"config":{"wifi":{"ssid":{"enable":true,"ssid":"NAME",
"pass":"PASS"}}}}}
{"id":2,"method":"Config.Save","params":{"reboot":true}}
```

## Cloud provisioning

1. Download `mos` tool by following Step 1 at <https://mongoose-os.com/docs/quickstart/setup.md>
2. Connect CCM to your workstation over a serial line

### AWS IoT provisioning

```
mos aws-iot-setup --aws-iot-region AWS-REGION
```

## Microsoft Azure provisioning

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

## Google IoT Core provisioning

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

## IBM Watson provisioning

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

## Factory reset

CCM provides a way to factory-reset the module. Factory reset clears WiFi configuration, and opens an AP after boot for further WiFi provisioning. Factory reset is triggered by the "long button press", i.e. setting the factory reset GPIO pin low and holding it in the low state for some time, defined by the timeout value.

See Configuration Reference section for the exact tunables.

## Security

CCM modules provide high level of security, employing several layers of protection:

- Flash encryption. Reading flash returns garbage. An attempt to copy the flash to clone CCM devices won't work.
- TLS 1.2 communication. Connections to the cloud services are secured by the modern TLS 1.2 standard.
- Ability to lock the configuration. Device vendor can choose the level of control end-user has over CCM - for example, completely disable any changes to configuration. By default, a CCM is provisioned to the cloud by vendor, and provisioned to WiFi by end-user.

## Configuration reference

CCM configuration is a JSON document that is stored on CCM's flash and could be managed by the following RPC functions:

- `Config.Get` - get configuration. Optional parameters: `{"key": "..."}` specifies the sub-tree to retrieve. If params is omitted, then the whole configuration tree is retrieved.
- `Config.Set` - set configuration. Parameters: `{"config": {...}}` specifies the sub-tree to change.
- `Config.Save` - saves configuration.

Below is a CCM configuration available for tuning:



```
{
  "provision": {
    "btn": {
      "pin": 0,           // Factory reset pin
      "pull_up": true,  // Pull-up state
      "hold_ms": 0      // Number of milliseconds to hold for reset
    },
    "led": {
      "pin": -1,        // Status LED GPIO number
      "active_high": true // LED is on when the voltage is high
    },
    "stable_state": 3,  // Do not reset when joined WiFi at least once
    "timeout": 300,
    "max_state": 3
  }
}
```

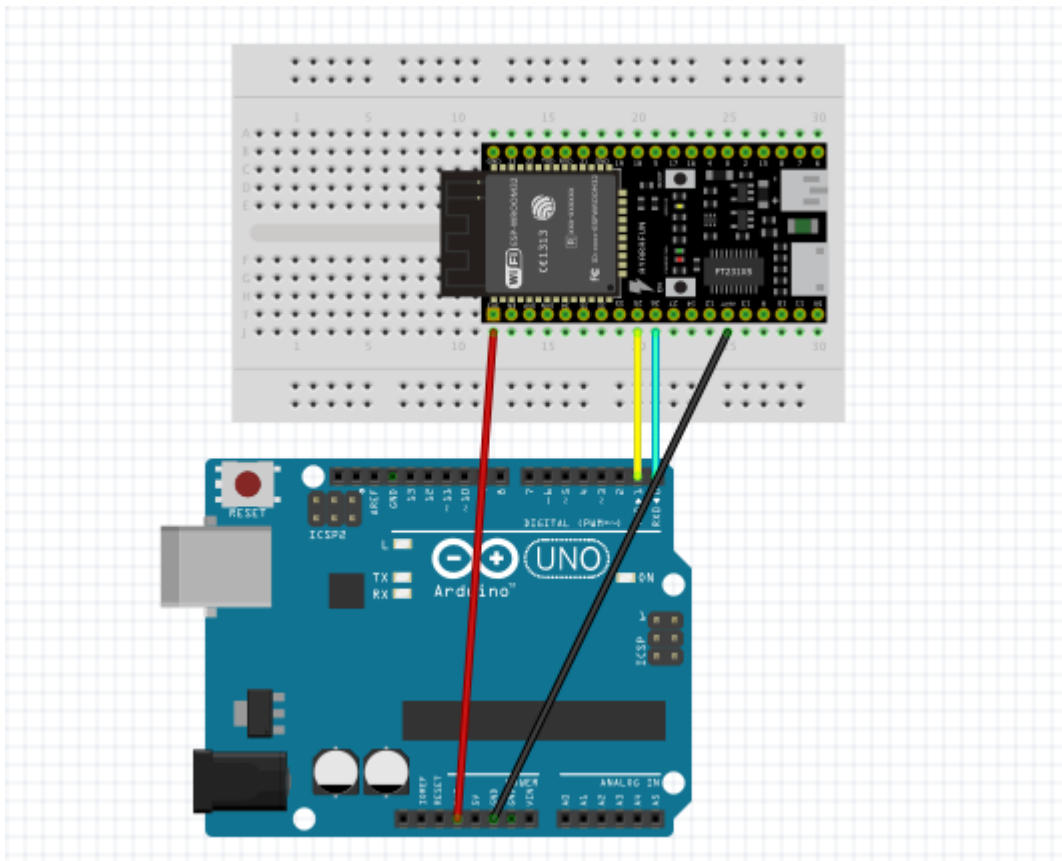
# CCM-EVAL evaluation board

The CCM-EVAL is an Espressif DevKit-C board with the CCM firmware loaded on it. The board uses CCM-MOD module, thus it works exactly the same way as CCM-MOD in a final design.

## Wiring

- Connect Host's RX to CCM pin 25
- Connect Host's TX to CCM pin 26
- Connect CCM pins GND and 3v3

Example of Arduino Uno wiring:



## Load Arduino sketch

Visit [shadow.ino](https://shadow.ino) Arduino sketch example. It demonstrates how to implement LED on/off via the device shadow. It works with any cloud that supports device shadow, for example AWS IoT, Azure IoT, [mDash](https://mDash.com).

See <https://mongoose-os.com/ccm/> for video tutorial.