МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Центральноукраїнський національний технічний університет

Смірнов В.В., Смірнова Н.В.

Програмування пристроїв Internet Of Things на базі мікроконтролера ESP8266 мовою програмування LUA

Навчальний посібник

Кропивницький - 2024

УДК: 004.41
ББК 32.97
С50

Рекомендовано Вченою радою Центральноукраїнського національного технічного університету до друку та використанню підручника у навчальному процесі здобувачами вищої освіти очної та заочної форми навчання.
Протокол № 9 від 27.05.2024 року

Рецензенти:

Мацуй Анатолій Миколайович, доктор технічних наук, професор кафедри автоматизації виробничих процесів Центральноукраїнського національного технічного університету.

Баранюк Олександр Филимонович, кандидат технічних наук, доцент кафедри інформатики та інформаційних технологій Центральноукраїнського державного університету імені Володимира Винниченка.

С50 **Смірнов В.В., Смірнова Н.В.**
Програмування пристроїв Internet Of Things на базі мікроконтролера ESP8266 мовою програмування LUA: навчальний посібник. – Кропивницький : ЦНТУ, 2024. – 417 с.

У навчальному посібнику подано базові відомості з мови програмування Lua, опис функцій API для програмування вбудованих модулів мікроконтролера ESP8266 таких, как Node, Wi-Fi, ADC, PWM, GPIO, UART, SPI, I2C, Timer, Crypto, SPIFFS.

Представлено опис мережевих функцій API для програмування мережевих модулів MQTT, HTTP, HTTPD, CoAP, TLS, WebSocket, OTA.

Представлено опис бездротової локальної мережі «Easy Net Everywhere» на базі якої мікроконтролери ESP8266 за допомогою інтерфейсу UART на локальному рівні об'єднуються для вирішення різноманітних завдань.

Описано конфігурацію різних моделей побудови мережі, їхні особливості та практичну реалізацію. Описано систему адресації вузлів мережі для простої та кластерної моделі мережі. Описано процес конфігурування, тестування та перевірки працездатності мережі.

Навчальний посібник призначений для здобувачів вищої освіти очної та заочної форми навчання за спеціальностями:

122 «Комп’ютерні науки»;
123 «Комп’ютерна інженерія»;
172 «Електронні комунікації та радіотехніка»;
151 «Автоматизація та комп'ютерно-інтегровані технології».

Навчальне електронне видання комбінованого використання.
Можна використовувати в локальному та мережному режимах

УДК: 004.41
ББК 32.97

© В.В. Смірнов, Н.В. Смірнова / 2024
© ЦНТУ / 2024

ЗМІСТ

ВСТУП 5

1. МОВА ПРОГРАМУВАННЯ LUA ДЛЯ КОНТРОЛЛЕРА ESP8266 . 8

2. API ПРОГРАМУВАННЯ ВСТРОЕННЫХ МОДУЛІВ ESP8266 39

Модуль wifi 40

Модуль wifi.sta 60

Модуль wifi.ap 81

Модуль wifi.ap.dhcp 90

Модуль wifi.eventmon 92

Модуль wifi.monitor 98

Модуль pwm 116

Модуль pwm2 121

Модуль uart 133

Порт softuart 140

Модуль таймера 142

Модуль fifo 153

Модуль i²c 157

Модуль spi 164

Криптографічний модуль 171

Модуль gpio 188

Файлова система spiffs 210

3. API ПРОГРАМУВАННЯ МЕРЕЖЕВИХ МОДУЛІВ ESP8266 235

Модуль node 235

Модуль node.egc 260

Модуль sjson 275

Модуль http 284

Модуль httpd 290

Модуль coap 299

Модуль tls 307

Модуль websocket 319

Модуль ggossip 325

Модуль оновлення ota 331

4. БЕЗДРОТОВА МЕРЕЖА "EASY NET EVERYWHERE" 338
Опис компонентів і вузлів мережі 344
Призначення та структура вузлів мережі.. 348
Архітектура, масштабування, адресація та робота мережі 353
Підготовка вузлів мережі до роботи 362
Тестування мережі. Передача команд і даних 368
Обмін даними в мережі. Приклади використання......................... 374
Робота з вузлом "controller" 377
AT-команди керування параметрами вузлів мережі 391

ВСТУП

На сьогодні розробка та програмування устройств «Internet Of Things» є актуальним завданням.

Технологія "Internet Of Things" у загальному випадку включає в себе:

- глобальну мережу Interne;

- мережеві протоколи Internet;

- мікроконтролери з вбудованими апаратними та мережевими програмними модулями;

- програмне забезпечення (API) для програмування вбудованих апаратних і програмних мережевих модулів;

- локальну бездротову мережу для організації взаємодії мікроконтролерів на локальному рівні.

Поширеною мовою програмування пристроїв "Internet Of Things" є LUA - скриптова мова програмування, розроблена в підрозділі Tecgraf (Computer Graphics Technology Group) Католицького університету Ріо-де-Жанейро (Бразилія).

Інтерпретатор мови є вільно розповсюджуваним, з відкритим вихідним кодом мовою програмування Сі.За можливостями, ідеологією і реалізацією, Lua найближча до JavaScript, проте Lua відрізняється могутнішими й набагато гнучкішими конструкціями.

Хоча Lua не містить поняття класу і об'єкта в явному вигляді, механізми об'єктноорієнтованого програмування (ООП) з підтримкою прототипів (включаючи множинне успадкування) легко реалізуються з використанням метатаблиць, які також дозволяють перевантаження операцій тощо.

Реалізована модель ООП (як і в JavaScript) - прототипна.

Lua отримала велике поширення в ролі вбудованої в інші проєкти мови сценаріїв (наприклад, для визначення конфігурації або для написання розширень). Lua комбінує простий процедурний синтаксис з потужними

можливостями опису даних через використання асоціативних масивів і розширюваної семантики мови.

У Lua використовується динамічна типізація, мовні конструкції перетворюються на байт-код, який виконується поверх регістрової віртуальної машини з автоматичним збирачем сміття.

Завдяки використанню функцій C Lua можна розширити, щоб справлятися з широким спектром різних доменів, таким чином створюючи налаштовані мови програмування, які спільно використовують синтаксичну структуру.

Важливою перевагою інтерпретатора є його здатність виконувати команди та скрипти "на льоту".

У навчальному посібнику подано базові відомості з мови програмування Lua, опис функцій API для програмування вбудованих і зовнішніх модулів мікроконтролера ESP8266, опис функцій API для програмування мережевих модулів і опис бездротової локальної мережі «Easy Net Everywhere» (рис.1).

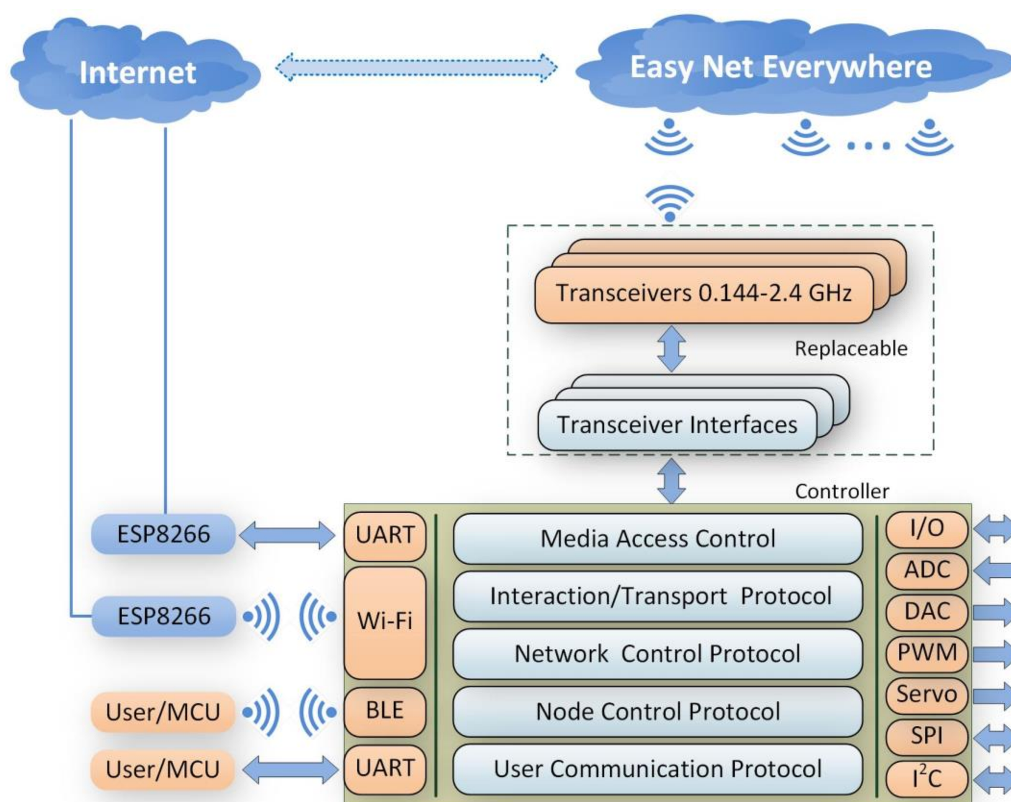


Рисунок 1 - Концепція "Internet Of Things" для контролера ESP8266

Завданням навчального посібника є навчання студентів навичкам роботи з мікроконтролером ESP8266 спільно з бездротовою локальною мережею «Easy Net Everywhere», а також для підготовки програмістів в області розробки та управління територіально-розподіленими системами, роботами, об'єктами і комплексами.

Навчальний посібник призначений для здобувачів вищої освіти очної та заочної форми навчання за спеціальностями:

122 «Комп’ютерні науки»;

123 «Комп’ютерна інженерія»;

172 «Електронні комунікації та радіотехніка»;

151 «Автоматизація та комп'ютерно-інтегровані технології».

1. МОВА ПРОГРАМУВАННЯ LUA ДЛЯ КОНТРОЛЛЕРА ESP8266

Контролер ESP8266 розроблено та виготовлено в Китаї компанією Espressif Systems.

Espressif також розробив і випустив додатковий комплект розробки програмного забезпечення (SDK), щоб дозволити розробникам створювати практичні програми IoT для ESP8266. SDK надається у вільний доступ для розробників у формі бінарних бібліотек і документації SDK.

Прошивка ESP8266 Lua є програмою ESP8266 і тому має бути накладена на ESP8266 SDK. Тим не менш, гачки та функції Lua дозволяють легко інтегрувати його без втрати будь-яких стандартних функцій мови Lua.

ESP8266 Lua базується на eLua, повнофункціональній реалізації Lua 5.1, яка була оптимізована для розробки та виконання вбудованих систем, щоб забезпечити структуру сценаріїв, яку можна використовувати для доставки корисних програм в межах обмежених ресурсів оперативної та флеш-пам'яті вбудованих процесорів, таких як ESP8266.

Однією з головних змін, внесених у форк eLua, є використання таблиць і констант, доступних лише для читання, де це можливо для бібліотечних модулів. У типовій збірці цей підхід зменшує обсяг оперативної пам'яті приблизно на 20-25 КБ, і це робить реалізацію Lua для ESP8266 можливою.

Основний вплив пакета SDK ESP8266 разом із обмеженнями його апаратних ресурсів полягає не в самій реалізації мови Lua, а в тому, як програмісти повинні підходити до розробки та структурування своїх програм.

Як детально описано нижче, SDK не є превентивним і керується подіями. Завдання можна пов'язувати з певними подіями за допомогою SDK API для реєстрації функцій зворотного виклику для відповідних подій. Події ставляться у внутрішню чергу SDK, а потім він викликає пов'язані завдання по одному, при цьому кожне завдання повертає керування SDK після завершення.

У SDK зазначено, що якщо будь-які завдання виконуються більше 15 мс, такі служби, як WiFi, можуть вийти з ладу.

Бібліотеки ESP8266 діють як обгортки C навколо зареєстрованих функцій зворотного виклику Lua, щоб дозволити їх використовувати як завдання SDK.

Тому необхідно використовувати стиль програмування, керований подіями, у написанні своїх програм ESP8266 Lua.

Більшість програмістів звикли писати в процедурному стилі, де існує чіткий єдиний потік виконання, а програма взаємодіє зі службами операційної системи за допомогою набору синхронних викликів API для здійснення мережевого вводу-виводу тощо.

ESP8266 використовує комбінацію вбудованої оперативної пам'яті та зовнішньої флеш-пам'яті, з'єднаних за допомогою спеціального інтерфейсу SPI. Код можна виконувати безпосередньо з адресного простору Flash-карти.

Насправді апаратне забезпечення ESP фактично виконує код в оперативній пам'яті, а у випадку флеш-відображених адрес воно виконує цей код із базованого на оперативній пам'яті кешу L1, який відображається на флеш-адреси.

Якщо адресований рядок знаходиться в кеші, то код виконується на повній тактовій частоті, але якщо ні, апаратне забезпечення прозоро обробляє помилку адреси, спочатку копіюючи код із флеш-пам'яті в оперативну пам'ять.

Це значною мірою прозоро з точки зору програмування додатків ESP8266, хоча помилковий доступ виконується на швидкості SRAM, і цей код працює, можливо, у 13 разів повільніше, ніж уже кешований код.

На відміну від розробки Arduino або ESP8266, де кожна зміна додатка потребує перепрошивання нової копії прошивки, у випадку Lua прошивка зазвичай прошивається один раз, і вся розробка програми виконується шляхом оновлення файлів у файловій системі SPIFFS.

У цьому відношенні розробка Lua на ESP8266 набагато більше схожа на розробку програм на більш традиційному ПК. Микропрограмне забезпечення буде оновлено, лише якщо розробник захоче додати або оновити одну чи кілька бібліотек, пов'язаних з обладнанням.

Різні бібліотеки (net, tmr, wifi тощо) використовують механізм зворотного виклику SDK, щоб прив'язати обробку Lua до окремих подій (наприклад, спрацьовування таймера таймера). Розробники повинні повністю використовувати ці події, щоб послідовності виконання Lua були короткими.

Обробка не за допомогою Lua (наприклад, мережеві функції) зазвичай відбувається лише після завершення виконання поточного блоку Lua. Тому будь-які мережеві виклики слід переглядати за асинхронним запитом.

Завдання не може почати оброблятися, доки код Lua не повернеться до виклику функції C, щоб дозволити цьому запущеному завданню вийти. Накопичування таких запитів в одній функції завдання Lua спалює дефіцитну оперативну пам'ять і може викликати panic event.

Це стосується таймера, мережі та інших зворотних викликів.

SDK використовує невелику кількість процедур обслуговування переривань (ISR) для обробки короткочасних критичних апаратних переривань, пов'язаних з обробкою. Вони дуже короткі й можуть перервати запущене завдання на термін до 10 мкс. (Зміна цих ISR або додавання нових не є життєздатним варіантом для більшості розробників.)

Уся інша обробка служб і програм розділена на блоки виконання коду, відомі як завдання. Окремі завдання виконуються по черзі та доводяться до кінця. Жодне завдання не може випередити інше.

Виконувані завдання ставляться в одну з трьох пріоритетних черг, а SDK містить простий планувальник, який виконує завдання в черзі FIFO в межах пріоритету.

Черга з високим пріоритетом використовується для завдань, пов'язаних з апаратним забезпеченням, середня для завдань, керованих таймером і подіями, а черга з низьким пріоритетом для всіх інших завдань.

Важливо, щоб час виконання завдань був максимально коротким, щоб уся система могла працювати безперебійно та швидко реагувати. Загальна рекомендація полягає в тому, щоб завдання із середнім пріоритетом не перевищували 2 мс, а завдання з низьким пріоритетом - не менше 15 мс.

Якщо завдання займають більше 500 мс, сторожовий таймер скине процесор. Цей сторожовий таймер можна скинути на рівні програми за допомогою функції tmr.wdclr(), але цього слід уникати.

Завдання програми можуть вимкнути переривання, щоб запобігти перериванню ISR критичної за часом частини коду.

SDK надає C API для взаємодії з ним; це включає набір функцій для оголошення функцій додатків (написаних на C) як зворотних викликів, щоб пов'язати завдання додатків із певним апаратним забезпеченням і подіями таймера, і їхнє виконання буде чергуватися із завданнями обробки SDK WiFi та Network.

По суті, мікропрограма ESP8266 - це програма на C, яка використовує здатність Lua виконуватися як вбудована мова та середовище виконання, щоб віддзеркалити цю структуру на рівні сценаріїв Lua. Усі складності та інтерфейс SDK і апаратного забезпечення загорнуті в бібліотеки мікропрограм, які перетворюють відповідні виклики у відповідний Lua API.

Під час завантаження SDK запускає перехоплювач у мікропрограмі. Цей код мікропрограми ініціалізує середовище Lua, а потім намагається запустити модуль Lua init.lua з файлової системи SPIFFS.

Потім цей init.lua модуль можна використовувати для виконання будь-якої необхідної ініціалізації програми та виклику необхідних нагадувань таймера або викликів бібліотеки для підпрограм прив'язки та зворотного виклику для виконання завдань, необхідних у відповідь на будь-які системні події.

За замовчуванням середовище виконання Lua також «слухає» UART 0, послідовний порт, в інтерактивному режимі та виконуватиме будь-які команди Lua, введені через цей послідовний порт.

Використання послідовного порту таким чином є найпоширенішим методом розробки та налагодження програм Lua на ESP8266.

Бібліотеки Lua надають набір функцій для оголошення функцій програми (написаних мовою Lua) як зворотних викликів (які зберігаються в реєстрі Lua), щоб пов'язати завдання програми з певним обладнанням і подіями таймера.

Вони також не є превентивними на рівні програм. Бібліотеки Lua працюють разом із SDK, щоб поставити в чергу незавершені події та викликати будь-які зареєстровані процедури зворотного виклику Lua, які потім виконуються безперервно.

Функції Lua, що виконуються надто довго (або фрагменти коду Lua, що виконуються в інтерактивній підказці через UART 0), можуть призвести до тайм-ауту інших системних функцій і служб або до розподілу обмежених ресурсів оперативної пам’яті для буферизації даних у черзі, що може викликати або сторожовий таймер, або виснаження пам’яті, обидва з яких призведуть до перезавантаження системи.

Подібно до своїх аналогів на C, завдання Lua, ініційовані таймером, мережею, GPIO та іншими зворотними викликами, виконуються без випередження до завершення перед виконанням наступного завдання, і це включає завдання SDK.

Друк на послідовний порт за замовчуванням виконується бібліотеками середовища виконання Lua, але служби SDK, включаючи навіть запит на перезавантаження, виконуються як окремі завдання.

Цей керований подіями підхід дуже відрізняється від звичайних процедурних програм, написаних на Lua.

Lua підтримує процедурний, об'єктний і функціональний стилі програмування, але є одночасно простою мовою.

Інтерпретатор Lua написаний на ANSI-C і являє собою бібліотеку, яку можна підключити до будь-якої програми.

У цьому випадку керуюча програма може викликати бібліотечні функції для виконання ділянки коду на Lua та роботи з даними, визначеними в цьому коді.

Також керуюча програма може зареєструвати власні функції таким чином, щоб їх можна було викликати з коду на Lua.

Остання можливість дозволяє використовувати Lua як мову, яку можна адаптувати до довільної області застосування.

Лексичні угоди

Ідентифікатори можуть містити букви, цифри та символи підчеркування і не можуть починатися з цифр.

Ідентифікатори, що починаються з підчеркування і складаються тільки з головних букв, зарезервовані для внутрішнього використання інтерпретатором.

В ідентифікаторах розрізняється верхній і нижній реєстри букв.

Строкові букви можна укласти в одинарні або подвійні кавычки. У них можна використовувати наступні спеціальні послідовності символів:

\n	переклад строки (LF = 0x0a)	\a	дзвоник
\r	повернення каретки (CR = 0x0d)	\b	Backspace
\t	табуляция	\f	форму подачі
\\	символ обратной косой черты	\v	вериткальная табуляция
\"	кавычка	\[	левая квадратная скобка
\'	апостроф	\]	правая квадратная скобка
\ddd	символ з кодом ddd (десятичним)	\0	символ з кодом 0

Якщо в кінці строки вихідного файлу міститься символ зворотної косої риси, то в наступному рядку може бути продовжено визначення строкового літералу, в якому в цьому місці вставляється символ нової строки.

Строкові букви можна також укласти в подвійні квадратні скобки [[….]]. У цьому випадку літерал може бути визначений на кількох строках (символи перекладу строки включаються в строковий літерал) і в ньому не інтерпретуються спеціальні послідовності символів.

Якщо безпосередньо після символів '[[' іде переклад строки, то він не включається в строковий літерал.

У якості обмежувачів рядка, крім подвійних квадратних скобок, може використовуватися символ [===[…]===], у якому між повторюваними квадратними скобами розташовано довільне число знаків рівності (одинакове для відкриваючого та закриваючого обмежувача).

У числових константах можна вказувати необов'язкову дробну частину і необов'язковий десятичний порядок, заданий символами 'e' або 'E'. Цілочисленні чисельні константи можна задавати в 16-річній системі, використовуючи префікс 0x.

Коментар починається символами ' -- ' (два мінуса підряд) і продовжується до кінця рядка. Якщо безпосередньо після символів ' -- ' ідуть символи '[[', то коментар є багатостроковим і продовжується до символів ']]'.

Багатострочний коментар може містити вкладені пари символів [[….]]. У якості обмежувачів багатострокових коментарів, крім подвійних квадратних скобок, також може використовуватися символ [===[…]===], у якому між повторюваними квадратними скобами розташовано довільне число знаків рівності (одинакове для відкриваючого та закриваючого обмежувача).

Строкова константа екранує символи початку коментаря.

Якщо перша строка файлу починається з символу '#', то вона пропускається. Це дозволяє використовувати Lua як інтерпретатор скриптів у Unix-подібних системах.

Типи даних

У Lua є такі типи даних:

nil	порожньо
boolean	логічний
number	числовий
string	рядковий
function	функція
userdata	користувацькі дані
thread	потік
table	асоціативний масив

Тип nil відповідає відсутності у змінному значенні. Цей тип відповідає єдиному значенню nil.

Логічний тип має два значення: істина і хибність.

Значення nil розглядається як false. Усі інші значення, включаючи число 0 і пусту строку, розглядаються як логічне значення true.

Усі числа представлені як реальні числа подвійної точності.

Строки являють собою масиви 8-бітних символів і можуть містити всередині себе символ з нульовим кодом. Усі строки в Lua постійні т.е. змінити содержимое існуючої строки не можна.

Функції можна використовувати змінним, передавати у функції в якості аргументу, повертати в якості результату з функцій і зберігати в таблицях.

Тип userdata відповідає нетипизованому указателю, за яким можуть бути розміщені довільні дані. Програма на Lua не може безпосередньо працювати з такими даними (створювати, модифікувати їх). Цей тип даних не відповідає жодним попередньо визначеним операціям, крім присвоєння та зрівняння на рівність. В той же час такі операції можуть бути визначені за допомогою механізму метаметодов.

Тип потоку відповідає незалежному виконуваному потоку. Цей тип даних використовується механізмом програми.

Тип таблиці відповідає таблицям - асоціативним масивам, які можна індексувати будь-якими значеннями та які можуть одночасно містити значення довільних типів.

Тип об'єкта, збереженого в змінній, можна пояснити, викликавши функцію type().

Ця функція повертає рядок, що містить канонічну назву типу: "nil", "number", "string", "boolean", "table", "function", "thread", "userdata".

Перетворення між числами і рядками відбуваються автоматично в момент їх у відповідному контексті використання.

Арифметичні операції розуміють числові аргументи і спроба виконати таку операцію над рядками приведе до перетворення їх у числа.

Строкові операції, вироблені над числами, приводять до їх перетворення в строку за допомогою іншого фіксованого форматного перетворення.

Можна також явно сформувати об'єкт у рядку за допомогою функції tostring() або в число за допомогою функції tonumber(). Для більшого контролю над процесом перетворення чисел у рядках слід використовувати функцію форматного перетворення.

Змінні

В Lua змінні не потрібно описувати. Перемінна з'являється в момент її першого використання. Якщо використовується змінна, яка не була попередньо ініціалізована, то вона має значення нуль. Перемінні не мають статичного типу, тип змінної визначається її поточним значенням.

Перемінна вважається глобальною, якщо вона явно не об'являється як локальна. Об'єднання локальних змінних може бути розташовано в будь-якому місці блоку і може бути спільно з їх ініціалізацією:

```
local x, y, z
local a, b, c = 1, 2, 3
local x = x
```

При ініціалізації локальної змінної справи від знака рівності введена змінна ще не доступна і використовується значення змінної, зовнішньої по відношенню до поточного блоку. Саме тому правильний приклад у третьому рядку (він демонструє часто використовувану ідіому мову).

Для локальних змінних інтерпретатор використовує лексичні області видимості, т.е. область дії змінної простирається від місця її опису (первого використання) і до кінця поточного блоку.

При цьому локальна змінна видима в блоках, внутрішніх по відношенню до блоку, в якому вона описана. Локальная переменная исчезает при виході з області видимості.

Якщо локальна змінна визначена поза блоком, то така змінна виконується за завершенням виконання цього ділянки коду, оскільки участок коду виконується інтерпретатором як безвимінна функція.

Ініціалізована глобальна змінна існує весь час функціонування інтерпретатора.

Для видалення змінної ей можна просто присвоїти значення nil.

Массиви, функції та дані користувача є об’єктами. Усі об’єкти анонімні і не можуть мати значення змінної.

Переменные сохраненные ссылки на объекты. При присваивании, передачі в функцію в якості аргументу і повернення з функції в якості результату не відбувається копіювання об'єктів, копіюються тільки посилання на них.

Таблиці

Таблиці (таблиця підказок) відповідають асоціативним масивам, які можна індексувати будь-якими значеннями крім nil і які можуть одночасно містити значення довільних типів крім nil. Елементи таблиці можна індексувати та об’єктами - таблицями, функціями та об’єктами типу userdata. Елементи масиву, які не отримали значення в результаті присвоєння, за замовчуванням мають значення нуль.

Таблиці - основна структура даних у Lua. З їх допомогою представлені також структури, класи та об'єкти. В цьому випадку використовується індексування строковим іменем поля структури.

Оскільки елементом масиву може бути функція, в структурах допускаються також і методи.

Для індексування масивів використовуються квадратні скобки: array[index].

Запис struct.field еквівалентна наступному запису: struct["field"].

Ця синтаксична особливість дозволяє використовувати таблиці в якості записів із позначеними полями.

Конструктор таблиці - це виображення, створення та повернення нової таблиці. Каждое виконання конструктора створює нову таблицю.

Конструктор таблиці являє собою закритий у фігурних скобках список ініціалізаторів полів (можливо порожніх), розділених зап'ятою або символом ';' (точка із запятою).

Для ініціалізаторів поля допустимі наступні варіанти:

```
[exp1] = exp2                 table[exp1] = exp2
name = exp                    table["name"] = exp
exp                           table[j] = exp
```

В останньому випадку змінна j пробігає послідовні цілі значення, починаючи з 1. Ініціалізатори перших двох елементів не змінюють значення цього лічильника. Ось приклад конструювання таблиці:

```
x = { len = 12, 11, 12, [123] = 1123 }
```

Після виконання такого оператора поля таблиці отримують такі значення:

```
x["len"] = x.len = 12
x[1] = 11
x[2] = 12
x[123] = 1123
```

Якщо останнім елементом списку ініціалізаторів є виклик функцій, повертаються значення функцій послідовно поміщаються в список ініціалізаторів.

Це поведінка можна змінити, завершивши виклик функції в круглі скобки. У цьому випадку з усіх повернених функцій використовується лише перше.

Після останнього ініціалізатора може вийти необов'язковий символ-розділитель полів ініціалізаторів (зап'ята або точка з зап'ятою).

Операції

Ниже перераховані основні операції:

-	смена знака
+ - * /	арифметика
^	возведение в ступінь
== ~=	равенство
<= > >=	порядок
not and or	логика
..	конкатенация строк
#	получение длины строки або масиву

При застосуванні арифметичних операцій строки, що мають числове значення, приводяться до нього. При конкатенації числових значень вони автоматично преобразуются в строки.

При сравнении на рівність не виробляється перетворення типів. Об'єкти різних типів завжди вважаються різними.

Відповідно "0" ~= 0, а при індексуванні a[0] і a["0"] відповідають різним ячейкам масиву. При зрівнянні на рівність/нерівність об'єктів виробляється порівняння ссылок на об'єкти. Равними надаються змінні, посилаючі на один і той же об'єкт.

При визначенні порядку типи аргументів повинні складатися, т.е. числа зрівнюються з числами, а строки - зі строками.

Відношення рівності і порядку завжди дають в результаті істинний або хибний т.е. логічне значення.

У логічних операціях nil розглядається як false, а всі інші значення, включаючи нулевое число і пусту строку - як true.

При підрахунку значень використовується коротка схема - другий аргумент вираховується тільки якщо це необхідно.

Діє наступна таблиця пріоритетів і асоціативності операцій:

[right]	^
[left]	not # -(unary)
[left]	* /
[left]	+ -
[left]	< > <= >= ~= ==
[right]	..
[left]	і
[left]	and

Логічні операції та пов'язані з ними ідіоми

Оператор не завжди повертає логічне значення, приймаючи аргумент довільного типу (при цьому тільки значення nil відповідає логічному значенню false, інші ж трактуються як true).

На відміну від нього оператори і і або завжди повертають один із своїх аргументів.

Оператор або повертає свій перший аргумент, якщо його значення відмінне від false і nil, а другий аргумент у протилежному випадку.

Оператор і повертає свій перший аргумент, якщо його значення дорівнює false або nil, і другий аргумент у протилежному випадку.

Така поведінка заснована на тому, що всі значення, відмінні від нуля, трактуються як істинні.

З цією поведінкою пов'язано кілька загальновживаних ідіом. У наступній таблиці наведена ідіоматична операція, а справа - відповідна звичайна запис:

```
x = x or v              if x == nil then x = v end
x = (e and a) or b      if e ~= nil then x = a else x = b end
```

Перша ідіома часто використовується для отримання неініціалізованої змінної змінної значення.

Вторая ідіома еквівалентна С'шному оператору x = e ? a, b (здесь вважається, що значення змінної a відмінно від nil).

Оператори

У Lua немає виділеної функції, з якої починається виконання програми. Інтерпретатор послідовно виконує оператори, які він отримує з файлу або з керуючої програми.

При цьому він попередньо компілює програму в подвійному представленні, яке також може бути збережено.

Любий блок коду виконується як анонімна функція, тому в ньому можна визначити локальні змінні і з нього можна повернути значення.

Оператори можна (но не обов'язково) розділяти символом ';'.

Допускається багаторазове присвоєння:

```
var1, var2 = val1, val2
```

При цьому виробляється вирівнювання - лишні значення відбрасуються, а змінним, іншим недостатнім значенням присваюється значення nil.

Усі виявлення, що входять у праву частину множинного присвоєння, вираховуються до самого присвоєння.

Якщо функція повертає кілька значень, то за допомогою множинного присвоєння можна отримати повертані значення:

```
 x, y, z = f();
 a, b, c, d = 5, f();
```

У загальному випадку, якщо в кінці списку значень, розташованого справа від знака присвоєння, знаходиться виклик функцій, то всі значення, що повертаються функціями, записуються в кінці списку значень.

Це поведінка можна змінити, завершивши виклик функції в круглі скобки. У цьому випадку з усіх повернених функцій використовується лише перше.

Ниже перечислены основные операторы:

```
do ... end

if ... then ... end
if ... then ... else ... end
if ... then ... elseif ... then ... end
if ... then ... elseif ... then ... else ... end

while ... do ... end
repeat ... until ...

for var = start, stop do ... end
for var = start, stop, step do ... end

return
return ...
break
```

Блок do … end перетворює послідовність операторів в один оператор і відкриває нову область видимості, в якій можна визначити локальні змінні.

В операторах if, while і repeat всі значення виражень, відмінні від false і nil трактуються як справжні.

Ось загальна форма запису оператора, якщо:

```
if ... then ... {elseif ... then ...} [else ...] end
```

Оператор return може не містити повернених значень або містити одне або кілька виражень (список).

Оскільки код блоку виконується як ананімна функція, значення, що повертається, може бути не тільки у функціях, але й у довільному блоку коду.

Оператори return і break повинні бути останніми операторами в блоці (тобто повинні бути або останніми операторами в блоці коду, або розпоряджатися безпосередньо перед словами end, else, elseif, until).

Всередині блоку необхідно використовувати ідіому do return end або do break end.

Оператор виконується для всіх значень змінного циклу, починаючи від початкового значення і закінчуючи кінцевим значенням, виключно.

Третє значення, якщо воно задано, використовується як крок зміни змінного циклу. Всі ці значення повинні бути числовими.

Вони вичисляються тільки одна раз перед виконанням циклу.

Перемінна цикл локальна в цьому циклі і не доступна поза його тілом. Значення змінного циклу не можна змінювати всередині тіла циклу.

Ось псевдокод, демонструючий виконання оператора для:

```
do
  local var, _limit, _step = tonumber(start), tonumber(stop),
tonumber(step)
  if not (var and _limit and _step) then error() end
  while (_step>0 and var<=_limit) or (_step<=0 and var>=_limit) do
    ...
    var = var + _step
  end
end
```

Функції

Визначення функції - це виконується вираз (конструктор функції), результатом якого є функція типу об’єкта:

```
f = function( ... ) ... end
```

У скобках розміщується список (можливо порожній) аргументів функцій. Список аргументів функцій може заканчуватися троеточием - у цьому випадку функція має змінне число аргументів. Між закриваючою скобкою і оператором end розміщується тело функції.

Для визначення функцій маються наступні короткі форми:

```
function fname( ... ) ... end            fname = function( ... ) ... end
local function fname( ... ) ... end      local fname = function( ... )
... end
function x.fname( ... ) ... end          x.fname = function( ... ) ...
end
function x:fname( ... ) ... end          x.fname = function( self, ... )
... end
```

У момент виконання функцій конструктора будується також замикання - таблиця всіх доступних у функціях і зовнішніх по відношенню до її локальних змінних.

Якщо функція передається як повертаємо значення, то вона зберігає доступ до всіх змінних, що знаходяться в її замиканні. При кожному виконанні функції конструктора будується нове замикання.

Визов функцій складається з посилань на функцію та укладається в круглі скобки списку аргументів (можливо пустого).

Ссылка на функцію може бути будь-яким виображенням, результатом вичислення якої є функція. Між ссылкой на службу і відкривається круглою скобкою не може бути перекладена строка.

Для виклику функцій маються наступні короткі форми:

```
f{...}          f({...})
f('...')        f'...'
f("")           f""
f([[...]])      f[[...]]
x:f(...)        x.f(x, ...)
```

У першому випадку єдиний аргумент є побудованим на лету таблиці, а в наступних трьох - строковий літералом.

В останньому випадку x використовується як таблиця, з якої витягується змінна f, яка є ссылкой на викликану функцію, і ця сама таблиця передається в функцію в якості першого аргументу.

Ця синтаксична особливість використовується для реалізації об'єктів.

При виклику функцій значення простих типів копіюються в аргументах за значенням, а для об’єктів в аргументах копіюються посилання.

При виклику функції виробляється вирівнювання числа аргументів - лишние значення відбрасуються, а аргументи, відповідним недоліком значення, отримують значення нуль.

Якщо в кінці списку параметрів є функції, що повертають кілька значень, то всі вони додаються в список аргументів. Це поведінка можна змінити, завершивши виклик функції в круглі скобки. У цьому випадку з усіх повернених функцій використовується лише перше.

Якщо функція має змінне число аргументів, то значення, не подані у відповідності ні одному з аргументів, можуть бути отримані при використанні спеціального імені «…».

Це ім'я веде себе як набір значень, які повертаються функціями, тобто. при його виявленні всередині виражень або з середини списку значень використовується тільки перше повертаемое значення.

Якщо ж ім’я «…» відображається в останній позиції списку значень (у конструкторах таблиць, при множинному застосуванні, у списку аргументів виклику функції та в операторі return), то всі повернуті значення поміщаються в кінці цього списку (правило доповнення списків).

Для надання такого повідомлення ім’я «…» може бути розміщено в круглих скобках. Перетворити набір значень у список можна за допомогою ідіом {...}.

Якщо функція приймає єдиний аргумент і трактує його як таблицю, елементи якої індексуються іменами формальних параметрів, то в цьому випадку фактично реалізується механізм виклику змінених аргументів:

```
function rename( arg )
  arg.new = arg.new or arg.old .. ".bak"
  return os.rename(arg.old, arg.new)
end

rename{ old = "asd.qwe" }
```

Відворот із функцій відбувається як при завершенні виконання її тіла, так і при виконанні оператора return.

Оператор return може повернути одне або кілька значень. Якщо в кінці списку повертаних значень є функції виклику, повертаючої кілька значень, то всі вони додаються в список аргументів.

Це поведінка можна змінити, завершивши виклик функції в круглі скобки. У цьому випадку з усіх повернених функцій використовується лише перше.

Якщо функція викликається як оператор, то всі повернуті значення знищуються.

Якщо функція викликається з виражень або з середини списку значень, то використовується тільки перше повертаемое значення.

Якщо ж функція викликається з останньої позиції списку значень (у конструкторах таблиць, при множинному застосуванні, у списку аргументів при виклику функцій і в операторі return), то всі повернуті значення поміщаються в кінці цього списку (правило доповнення списків).

Для подання таких поведень виклик функції можуть бути розміщені в круглих скобках.

Мається попередньо визначена функція unpack(), яка приймає масив, елементи якого проіндексовані 1, і повертає всі його елементи.

Ця функція може бути використана для виклику функцій, які приймають змінне число аргументів з динамічним формуванням списку фактичних аргументів. Зворотна операція виконується таким чином:

```
list1 = {f()}  -- створити список з усіх значень, повернутих функцією f()
list2 = {...}  -- створити список з усіх значень, переданих у функцію
                  зі змінним числом аргументів
```

Перемінне число повертаних значень, правило доповнення списків і можливість передавати в функцію не всі формальні параметри можуть взаємодіяти нетривіальним способом.

Часто функція (наприклад, foo()) повертає відповідь при нормальному завершенні та нуль і повідомлення про помилку при ненормальному.

Функція assert(val, msg) генерує помилку з повідомленням повідомлення (викликає функцію error(msg)), якщо val має значення false або nil і повертає значення val у протилежному випадку. Тоді оператор

```
v = assert(foo(), "message")
```

у випадку успіху присваює змінну в значення, що повертається функцією foo().

У цьому випадку foo() повертає одне значення, а assert() отримує параметр msg, рівний nil. У разі помилки функція assert() отримує нуль і повідомляє про помилку.

Ітератори

Ітератори використовуються для перерахування елементів довільних послідовностей:

```
for v_1, v_2, ..., v_n in explist do ... end
```

Число змінних у списку v_1, …, v_n може бути довільним і не зобов’язане відповідати числу виражених у списку explist.

У ролі explist зазвичай виступає виклик функції-фабрики ітераторів. Така функція повертає функцію-ітератор, стан і початкове значення керуючої змінної циклу.

Ітератор інтерпретується таким чином:

```
do
  local f, s, v_1 = explist
  local v_2, ... , v_n
  while true do
    v_1, ..., v_n = f(s, v_1)
    if v_1 == nil then break end
    ...
  end
end
```

На кожному кроці значення всіх змінних вік вираховуються шляхом виклику функції-ітератора.

Значення керуючої змінної v_1 управляє завершенням циклу - цикл завершується як тільки функція-ітератор повертає нуль як значення для змінної var_1.

Фактично ітераціями керує змінна v_1, а інші змінні можна розглядати як «хвост», що повертається функцією-ітератором:

```
do
  local f, s, v = explist
  while true do
    v = f(s, v)
    if v == nil then break end
    ...
  end
end
```

Оператор, у якому викликається функція-фабрика, інтерпретується як звичайний оператор присвоєння т.е. функція-фабрика може повертати довільну кількість значень.

Ітератори без внутрішнього стану

Ітератор без внутрішнього стану не зберігає жодної внутрішньої інформації, яка дозволяє йому визначити своє положення в ітерованому контейнері.

Наступне значення керуючої змінної вираховується безпосередньо за її попереднім значенням, а стан використовується для зберігання посилань на ітерований контейнер.

Ось приклад простого ітератора без внутрішнього стану:

```
function iter( a, i )
  i = i + 1
  local v = a[i]
  if v then
    return i, v
  else
    return nil
  end
end

function ipairs( a )
  return iter, a, 0
end
```

Ітератори, зберігаючий стан в замиканні

Якщо ітератору для обходу контейнера необхідний внутрішній стан, то простіше всього зберігати його в замиканні, створюваному за контекстом функції-фабрики. Ось простий приклад:

```
function ipairs( a )
  local i = 0
  local t = a

  local function iter()
```

```
    i = i + 1
    local v = t[i]
    if v then
      return i, v
    else
      return nil
    end
  end

  return iter
end
```

Тут ітератор зберігає весь контекст у замиканні і не потребує в стані та поточному значенні керуючої змінної.

Відповідно, ітератор не приймає стан і керовану змінну, а фабрика не повертає значення стану і початкове значення керованої змінної.

Стандартные итераторы

Частіше всього ітератори застосовуються для обходу елементів таблиці. Для цього існує кілька заздалегідь визначених функцій-фабрик ітераторів. Фабрика pairs(t) повертає ітератор, що дає на кожному кроці індекс у таблиці та розміщене за цим індексом значення:

```
for idx, val in pairs(tbl) do
  ...
end
```

На самому ділі цей ітератор легко визначити, використовуючи стандартну функцію next(tbl, idx):

```
function pairs(tbl)
  return next, tbl, nil
end
```

Функція next(tbl, idx) повертає наступне для idx значення індексу за деяким обходом таблиці tbl (визов next(tbl, nil) повертає початкове значення індексу; після вилучення елементів таблиці повертається нуль).

Фабрика ipairs(tbl) повертає ітератор, який працює абсолютно аналогічно описаному вище, але призначений для обходу таблиці, проіндексованої цілими числами, починаючи з 1.

Мета-таблицы

Каждая таблиця і об’єкт типу userdata можуть мати мета-таблицю - звичайну таблицю, яка визначає поведінку вихідного об’єкта при застосуванні до нього деяких спеціальних операцій.

Наприклад, коли об’єкт виявляється про операнд при складанні, інтерпретатор ищет поле мета-таблиці з іменем __add і, якщо таке поле присутнє, щоб використовувати його значення як функцію, що виконує розташування.

Мета-таблиці дозволяють визначити поведінку об'єкта при арифметичних операціях, порівняннях, конкатенаціях та індексації.

Також можна визначити функцію, що викликається при звільненні об'єкта типу userdata.

Індекси (імена полей) в мета-таблиці називаються подіями, а відповідні значення (обробники подій) - метаметодами.

За умовчанням знову створена таблиця не має мета-таблиці.

Любу таблицю mt можна зробити мета-таблицею таблиці t, викликавши функцію setmetatable(t, mt).

Функція getmetatable(t) повертає мета-таблицю таблиці t або nil, якщо таблиця не має мета-таблиці.

Люба таблиця може виконувати роль мета-таблиці для будь-якої іншої таблиці, в тому числі і для себе.

Lua визначає наступні події:

__add, __sub, __mul, __div	арифметичні операції
__pow	возведение в ступінь
__ unm	унарный минус
__ concat	конкатенация
__ eq, __ lt, __ le	операции сравнения
__index	доступ по наявному індексу
__newindex	присвоение нового елементу таблицы
__call	виклик функції
__tostring	перетворення в строку
__metatable	отримання мета-таблиці

Вираз a ~= b вичисляється як ні (a == b). Вираз a > b вичисляється як b < a. Вираз a >= b вичисляється як b <= a.

При відсутності метаметода __le операція <= вичисляється як не (b < a) т.е. з допомогою метаметода __lt.

Для бінарних операцій такий вибір обробника виконується таким чином: запитується перший операнд і, якщо він не визначає обробник, то запитується другий операнд.

Для операцій порівняння метаметод вибирається тільки в тому випадку, якщо порівнювані операнди мають одинаковий тип і одинаковий метаметод для виконання цієї операції.

У руководстві користувача наведено псевдокод на Lua, демонструючий контекст виклику метаметодів.

Обробник події __index може бути функцією або таблицею.

У разі, якщо функції обработчик викликаються і йому передається таблиця і значення індексу.

Така функція повинна повернути результат індексування. У випадку таблиці відбувається повторне індексування цієї таблиці з індексом.

Якщо присутній обробіток події __newindex, то він викликається замість отримання значень нового елемента таблиці.

Якщо цей обробник є таблицею, то присвоєння виробляється в цій таблиці.

Метаметод __tostring дозволяє опрацювати перетворення об'єкта (таблиці або даних користувача) у рядку.

Метаметод __metatable дозволяє опрацювати операцію отримання мета-таблиці.

Якщо в цьому полі в мета-таблиці встановлено значення, функція getmetatable() поверне значення цього поля, функція setmetatable() завершиться з помилкою.

Функція rawget(tbl, idx) дозволяє читати поле таблиці в обхід механізму мета-методів.

Аналогічний доступ до запису надає функцію rawset(tbl, idx, val). Функція rawequal(t1, t2) порівнює об'єкти в обхід механізму мета-методів.

Приклади використання мета-таблиць

Значення за умовчанням для поля таблиці

У наступному прикладі мета-таблиці використовуються для призначення значень за замовчуванням для відсутніх елементів таблиці:

```
function set_def(t, v)
  local mt = { __index = function() return v end }
  setmetatable(t, mt)
end
```

Це більш нетривіальне рішення, яке не використовує окремі мета-таблиці для кожного зменшуваного значення:

```
local key = {}
local mt = { __index = function(t) return t[key] end }

function set_def(t, v)
  t[key] = v
```

```
  setmetatable(t, mt)
end
```

Тут локальний (пустий) ключ таблиці використовується як завідомо унікальний індекс, за яким у вихідній таблиці t зберігається зменшуване значення d.

Усі таблиці, для яких встановлюється зменшене значення відсутніх полів, розділяють загальну мета-таблицю mt.

Мета-метод __index цієї мета-таблиці перехоплює звернення до наявних полів таблиці та повертає значення, збережене в самій таблиці за індексом ключа.

У цьому рішенні є недолік: у таблиці з'являється нова пара ключ-значення, яке проявляється при спробі елементів знайти всі таблиці.

Таблиця-проксі

У наступному прикладі пуста таблиця виконує роль проксі, переадресуючого звернення до поля таблиці:

```
local key = {}
local mt = {
  __index = function(t,k)
    return t[key][k]
  end,
  __newindex = function(t,k,v)
    t[key][k] = v
  end
}

function proxy(t)
  local proxy = {}
  proxy[key] = t
  setmetatable(proxy, mt)
  return proxy
end
```

Тут локальний (пустий) ключ таблиці використовується як завідомо унікальний індекс, за яким у таблиці-проксі зберігається посилання на вихідну таблицю t.

Проксі представляє собою таблицю, єдиний елемент, який має ключ індексу, тому звернення до будь-якого елементу проксі приводиться до виклику мета-методу.

Загальна для всіх проксі мета-таблиця визначає мета-методи __index і __newindex, що захоплюють вихідну таблицю з єдиного елемента проксі, індексуючи його ключ таблиці.

Мета-методи можуть забезпечити довільну дисципліну обробки звернень до полів вихідної таблиці.

Простими прикладами є протоколювання звернень або генерація помилок при спробі змінити значення елемента таблиці.

"Слабкі" таблиці

Якщо об’єкт був використаний як індекс таблиці або посилання на нього було збережено в таблиці, то збірник мусора не зможе використовувати такий об’єкт.

У той же час, в деяких випадках бажано мати таблицю, в якій зв’язуються її елементи з ключами та/або значеннями «слаба» тощо. не препятствующая сбору мусора.

Зазвичай така необхідність виникає при кешуванні результатів, вичислених у таблиці, і збереження атрибутів об'єктів у таблиці, проіндексованій самими об'єктами.

У першому випадку бажана слабка зв'язок для значень, у другому - для індексів.

Зв'язок елементів таблиці з об'єктами (значеннями та індексами) визначає значення строкового поля __mode її мета-таблиці.

Якщо це поле містить символ 'k', то слабко робиться зв'язок для індексів (ключей); якщо він містить символ 'v', то слабко робиться зв'язок для значення. Поле може містити оба символи, які роблять слабкий зв'язок як для індексів, так і для значень.

Якщо використовувати слабкі таблиці, щоб для розглянутої вище задачі скласти таблиці зменшуваного значення для відсутніх полів, можна привести таке рішення:

```
local defaults = {}
setmetatable(defaults, { __mode = "k" })
local mt = { __index = function(t) return defaults[t] end }

function set_def(t, d)
  defaults[t] = d
  setmetatable(t, mt)
end
```

Ось інше рішення, в якому слабка таблиця зберігає мета-таблиці, кількість яких входить до числа різних зменшуваних значень:

```
local metas = {}
setmetatable(metas, { __mode = "v" })

function set_def(t, d)
  local mt = metas[d]
  if mt == nil then
    mt = { __index = function () return d end }
    metas[d] = mt
  end
  setmetatable(t, mt)
end
```

Глобальний контекст

Усі глобальні змінні є полями звичайної таблиці, яка називається глобальним контекстом.

Ця таблиця доступна через глобальну змінну _G. Оскільки всі глобальні змінні є полями контексту, то _G._G == _G.

Глобальний контекст робить можливим доступ до глобальних змінних за динамічно генерованим іменем:

```
val = _G[varname]
_G[varname] = val
```

Оскільки глобальний контекст є звичайною таблицею, йому може відповідати мета-таблиця. У наступному прикладі змінні середовища вводяться в глобальну область видимості як глобальні змінні, доступні тільки для читення:

```
local f = function (t,i)
  return os.getenv(i)
end

setmetatable(_G, {__index=f})
```

Цей же прийом дозволяє запобігти доступ до неініціалізованих глобальних змінних.

Пакети

Пакети - основний спосіб визначити набір взаємозв'язаних функцій, не загрязняючи при цій глобальній області видимості.

Звичайний пакет являє собою окремий файл, який у глобальній області видимості визначає єдину таблицю, що містить усі функції цього пакета:

```
my_package = {}

function my_package.foo()
  ...
end
```

Також можна робити всі локальні функції та окремо формувати таблицю експортованих функцій:

```
local function foo() ... end
local function bar() ... end

my_package = {
  foo = foo,
  bar = bar,
}
```

Пакет завантажується за допомогою функції require(), при чому під час завантаження імені, передане цю функцію (воно може не містити розширення, яке додається автоматично) доступно через змінну _REQUIREDNAME:

```
if _REQUIREDNAME == nil then
  run_some_internal_tests()
end
```

Класи і об'єкти

Конструкція tbl:func() (при об'явленні функцій і при її виклику) надає основні можливості, що дозволяють працювати з таблицями як з об'єктом. Основна проблема полягає в пошкодженні багатьох об'єктів, які мають сходну поведінку т.е. порожденных от одного класу:

```
function class()
  cl = {}
  cl.__index = cl            -- cl буде використовуватися як мета-таблиця
  return cl
end

function object( cl, obj )
  obj = obj or {}            -- можливо вже є заповнені поля
  setmetatable(obj, cl)
  return obj
end
```

Тут клас функції створює пусту таблицю, підготовлену до того, щоб стати мета-таблицею об'єкта.

Методы класса делаются полями этой таблицы т.е. клас є таблицею, що одночасно містить методи об'єкта та його мета-методи.

Функція object() створює об’єкт заданого класу - таблицю, в якій у якості мета-таблиці встановлено заданий клас.

У другому аргументі може бути передана таблиця, що містить проініціалізовані поля об'єкта.

```
Some_Class = class()
function Some_Class:foo() ... end
function Some_Class:new()
  return object(self, { xxx = 12 })
end
x = Some_Class:new()
x:foo()
```

Наслідування

В описаній реалізації мета-таблиця класу залишається не використаною, що робить просту задачу реалізації наслідування.

Клас-наслідник створюється як об'єкт класу, після чого в ньому встановлюється поле __index таким чином, щоб його можна було використовувати як мета-таблицю:

```
function subclass( pcl )
  cl = pcl:new()           -- створюємо екземпляр
  cl.__index = cl          -- і робимо його класом  return cl
end
```

Тепер в отриманий клас-наслідник можна додати нові поля і методи:

```
Der_Class = subclass(Some_Class)

function Der_Class:new()
  local obj = object(self, Some_Class:new())
  obj.yyy = 13                      -- додаємо нові поля
  return obj
end

function Der_Class:bar() ... end    -- і нові методи
y = Der_Class:new()

y:foo()
y:bar()
```

Єдиний нетривіальний момент тут полягає у використанні функції new() із класу-предка з наступною заміною мета-таблиці через виклик функції object().

При зверненні до методів об'єкта класу-наслідника в першу чергу відбувається їх пошук у мета-таблиці т.е. в самому класі-насліднику. Якщо метод був унаслідований, то цей пошук виявиться невдалим і виникне звернення до мета-таблиці класу-наслідника т.е. к классу-предку.

Основний недолік наведеного загального рішення полягає в неможливості передачі у функцію new() параметрів класу-предка.

Аргументи командної строки

Передані при запуску аргументи командного рядка доступні як елементи масиву arg.

2. API ПРОГРАМУВАННЯ ВСТРОЕННЫХ МОДУЛІВ ESP8266

Найбільш надійний і ефективний підхід до кодування додатків ESP8266 Lua полягає в тому, щоб прийняти цю парадигму моделі подій і розкласти програму на атомарні завдання, які об'єднані подіями, які самі ініціюють функції зворотного виклику.

Кожне завдання події встановлюється зворотним викликом у виклику API у попередньому завданні.

Зворотні виклики SDK включають:

Модуль Lua	Функції, які визначають або видаляють зворотні виклики
tmr	register([id,] interval, mode, function())
node	task.post([task_priority],function),output(function(str), serial_debug)
wifi	startsmart(chan, function()),sta.getap(function(table))
net.server	sk:listen(port,[ip],function(socket))
net	sk:on(event, function(socket, [, data])), sk:send(string, function(sent)), sk:dns(domain, function(socket,ip))
gpio	trig(pin, type, function(level))
mqtt	client:m:on(event, function(conn[, topic, data])
uart	uart.on(event, cnt, [function(data)], [run_input])

МОДУЛЬ WIFI

Підсистема WiFi підтримується фоновими завданнями, які повинні виконуватися періодично. Будь-яка функція або завдання, що займає більше 15 мс (мілісекунд), може призвести до збою підсистеми WiFi.

Щоб уникнути цих потенційних збоїв, рекомендується призупинити підсистему WiFi за допомогою wifi.suspend() перед виконанням будь-яких завдань або функцій, які перевищують ці рекомендації в 15 мс.

Режими WiFi

Пристрої, які підключаються до мережі WiFi, називаються станціями (STA). Підключення до Wi-Fi забезпечує точка доступу (AP), яка діє як концентратор для однієї або кількох станцій.

Точка доступу на іншому кінці підключена до дротової мережі. Точка доступу зазвичай інтегрована з маршрутизатором для забезпечення доступу з мережі Wi-Fi до Інтернету.

Кожна точка доступу розпізнається за SSID (Service S et ID entifier), що, по суті, є назвою мережі, яку ви вибираєте під час підключення пристрою (станції) до WiFi.

Кожен модуль ESP8266 може працювати як станція, тому ми можемо підключити його до мережі WiFi.

Він також може працювати як точка програмного доступу (soft-AP) для встановлення власної мережі WiFi. Тому ми можемо підключати інші станції до таких модулів.

По-третє, ESP8266 також може працювати як у режимі станції, так і в режимі програмної точки доступу одночасно. Це пропонує можливість побудови, наприклад, сітчастих мереж.

Станція

Режим станції (STA) використовується для підключення ESP8266 до мережі WiFi, створеної через точку доступу.

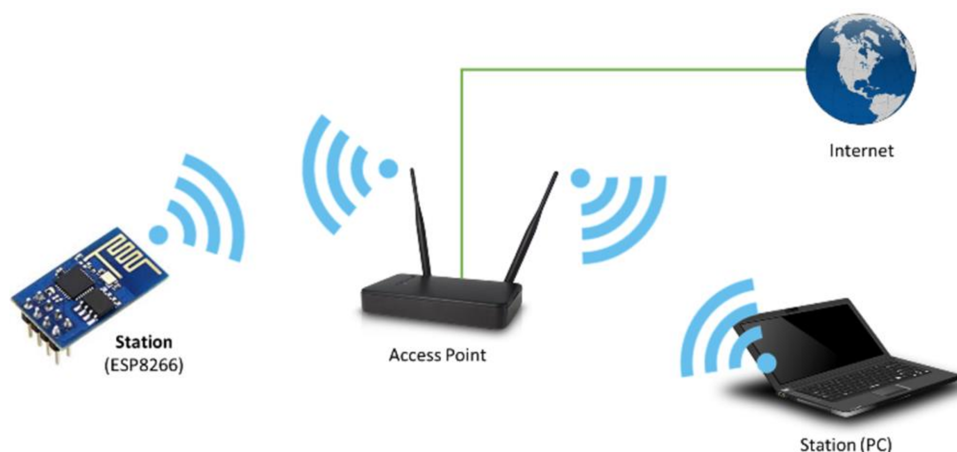


Рисунок 1 – Access point

Точка доступу (AP) - це пристрій, який надає доступ до мережі Wi-Fi іншим пристроям (станціям) і підключає їх далі до дротової мережі. ESP8266 може надавати подібні функції, за винятком того, що він не має інтерфейсу до дротової мережі.

Такий режим роботи називається програмною точкою доступу (soft-AP). Максимальна кількість станцій, підключених до програмної точки доступу, становить п’ять.

Рисунок 2 – Soft Access point

Режим soft-AP часто використовується і є проміжним кроком перед підключенням ESP до WiFi у станційному режимі.

Це коли SSID і пароль до такої мережі не відомі заздалегідь.

Модуль спочатку завантажується в режимі soft-AP, тому ми можемо підключитися до нього за допомогою ноутбука або мобільного телефону. Тоді ми можемо надати облікові дані цільовій мережі. Після цього ESP перемикається в станційний режим і може підключатися до цільової мережі WiFi.

Такі функції надає модуль налаштування кінцевого користувача ESP8266.

Станція + Soft Access Point

Іншим зручним застосуванням режиму програмної точки доступу є налаштування сітчастих мереж. ESP може працювати як у режимі програмної точки доступу, так і в режимі станції, тож він може діяти як вузол сітчастої мережі.

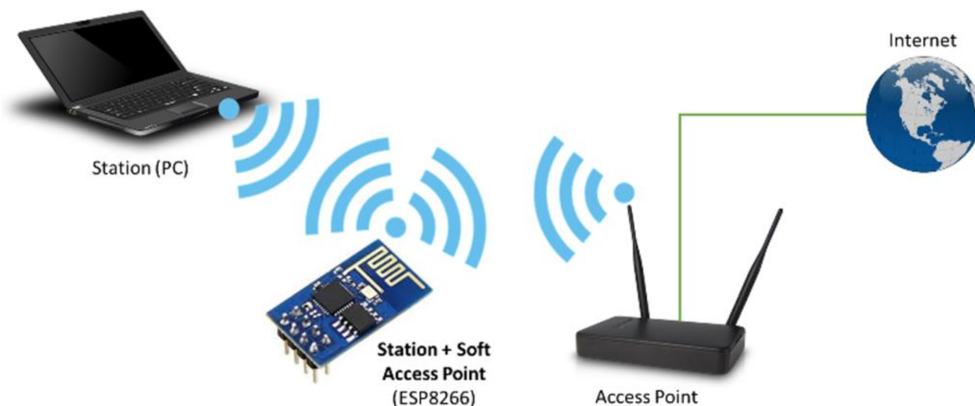


Рисунок 2 – Station +Soft Access point

Керування ESP8266 WiFi

Елемент керування ESP8266 WiFi розподілено на кілька таблиць:

wifi для загальної конфігурації WiFi

wifi.sta для функцій режиму станції

wifi.ap для функцій бездротової точки доступу (WAP або просто AP).

wifi.ap.dhcp для керування сервером DHCP

wifi.eventmon для моніторингу подій Wi-Fi

wifi.monitor для режиму моніторингу Wi-Fi

wifi.getchannel()	Отримує поточний канал WiFi.
wifi.getcountry()	Отримує поточну інформацію про країну.
wifi.getdefaultmode()	Отримує стандартний режим роботи WiFi.
wifi.getmode()	Отримує режим роботи WiFi.
wifi.getphymode()	Отримує фізичний режим WiFi.
wifi.nullmodesleep()	Налаштовує, чи WiFi автоматично переходить у сплячий режим у NULL_MODE.
wifi.resume()	Відключає Wi-Fi із призупиненого стану
wifi.setcountry()	Дає інформацію про поточну країну.
wifi.setmode()	Налаштовує режим WiFi для використання.
wifi.setphymode()	Встановлює фізичний режим WiFi.
wifi.setmaxtxpower()	Встановлює максимальну потужність передачі WiFi.
wifi.startsmart()	Починає автоматичне налаштування, у разі успіху автоматично встановлює SSID і пароль.
wifi.stopsmart()	Зупиняє процес розумного налаштування.
wifi.suspend()	Призупиняє Wi-Fi, щоб зменшити поточне споживання.
wifi.sta.autoconnect()	Автоматичне підключення до точки доступу в режимі станції.
wifi.sta.changeap()	Встановлює точку доступу зі списку, який повертає wifi.
wifi.sta.clearconfig()	Очищає поточну збережену конфігурацію станції WiFi, видаляючи її з флеш-пам’яті.
wifi.sta.config()	Встановлює конфігурацію станції WiFi.
wifi.sta.connect()	Підключає до налаштованої точки доступу в режимі станції.
wifi.sta.disconnect()	Відключає від точки доступу в режимі станції.

wifi.sta.getap()	Сканує список точок доступу як таблицю Lua у функцію зворотного виклику.
wifi.sta.getapindex()	Отримує індекс поточної точки доступу, що зберігається в кеші AP.
wifi.sta.getapinfo()	Отримує інформацію про точки доступу, кешовані станцією ESP8266.
wifi.sta.getbroadcast()	Отримує широкомовну адресу в станційному режимі.
wifi.sta.getconfig()	Отримує конфігурацію станції WiFi.
wifi.sta.getdefaultconfig()	Отримує стандартну конфігурацію станції Wi-Fi, збережену у флеш-пам’яті.
wifi.sta.gethostname()	Отримує назву хоста поточної станції.
wifi.sta.getip()	Отримує IP-адресу, маску мережі та адресу шлюзу в режимі станції.
wifi.sta.getmac()	Отримує MAC-адресу в режимі станції.
wifi.sta.getrssi()	Отримує RSSI (індикатор потужності отриманого сигналу) точки доступу, до якої підключилася станція ESP8266.
wifi.sta.setaplimit()	Встановлює максимальну кількість точок доступу для зберігання у флеш-пам’яті.
wifi.sta.sethostname()	Встановлює назву хоста станції.
wifi.sta.setip()	Встановлює IP-адресу, маску мережі, адресу шлюзу в режимі станції.
wifi.sta.setmac()	Встановлює MAC-адресу в станційному режимі.
wifi.sta.sleeptype()	Налаштовує тип сну WiFi-модему, який буде використовуватися, коли станція підключена до точки доступу.
wifi.sta.status()	Отримує поточний статус у станційному режимі.
wifi.ap.config()	Встановлює SSID і пароль у режимі AP.

wifi.ap.deauth()	Припиняє роботу (примусово видаляє) клієнта з точки доступу ESP, надсилаючи відповідний IEEE802.
wifi.ap.getbroadcast()	Отримує широкомовну адресу в режимі AP.
wifi.ap.getclient()	Отримує таблицю клієнтів, підключених до пристрою в режимі точки доступу.
wifi.ap.getconfig()	Отримує поточну конфігурацію SoftAP.
wifi.ap.getdefaultconfig()	Отримує стандартну конфігурацію SoftAP, збережену у флеш-пам'яті.
wifi.ap.getip()	Отримує IP-адресу, маску мережі та шлюз у режимі AP.
wifi.ap.getmac()	Отримує MAC-адресу в режимі AP.
wifi.ap.setip()	Встановлює IP-адресу, маску мережі та адресу шлюзу в режимі AP.
wifi.ap.setmac()	Встановлює MAC-адресу в режимі AP.
wifi.ap.dhcp.config()	Налаштовує службу dhcp.
wifi.ap.dhcp.start()	Запускає службу DHCP.
wifi.ap.dhcp.stop()	Зупиняє службу DHCP.
wifi.eventmon.register()	Реєстрація/скасування реєстрації зворотних викликів для монітора подій WiFi.
wifi.eventmon.unregister()	Скасувати реєстрацію зворотних викликів для монітора подій WiFi.
wifi.eventmon.reason	Таблиця з причинами відключення.

ОПИС ФУНКЦІЙ API МОДУЛЯ WIFI

wifi.getchannel()

Отримує поточний канал WiFi.

Синтаксис

`wifi.getchannel()`

Параметри

`nil`

Повернення

поточний канал WiFi

wifi.getcountry()

Отримує поточну інформацію про країну.

Синтаксис

`wifi.getcountry()`

Параметри

`nil`

Повернення

- `country_info` ця таблиця містить інформацію про поточну країну
  - `country` Код країни, рядок із 2 символів.
  - `start_ch` Стартовий канал.
  - `end_ch` Завершення каналу.
  - `policy` Параметр політики визначає, яку конфігурацію інформації про країну використовувати, інформацію про країну, надану станції точкою доступу, або локальну конфігурацію.

- 0 Політика країни є автоматичною, ESP8266 використовуватиме інформацію про країну, надану точкою доступу, до якої підключено станцію.
- 1 Політику країни встановлено вручну, ESP8266 використовуватиме локально налаштовану інформацію про країну.

приклад

```
for k, v in pairs(wifi.getcountry()) do
  print(k, v)
end
```

Дивись також

wifi.setcountry()

wifi.getdefaultmode()

Отримує стандартний режим роботи WiFi.

Синтаксис

wifi.getdefaultmode()

Параметри

nil

Повернення

Режим Wi-Fi як один із, wifi.STATION або

констант. wifi.SOFTAPwifi.STATIONAPwifi.NULLMODE

Дивись також

wifi.getmode() wifi.setmode()

wifi.getmode()

Отримує режим роботи WiFi.

Синтаксис

`wifi.getmode()`

Параметри

`nil`

Повернення

Режим Wi-Fi як один із, `wifi.STATION` або

констант. `wifi.SOFTAPwifi.STATIONAPwifi.NULLMODE`

Дивись також

`wifi.getdefaultmode()` `wifi.setmode()`

wifi.getphymode()

Отримує фізичний режим WiFi.

Синтаксис

`wifi.getphymode()`

Параметри

немає

Повернення

Поточний фізичний режим як

один `wifi.PHYMODE_B` із `wifi.PHYMODE_G` або `wifi.PHYMODE_N`.

Дивись також

`wifi.setphymode()`

wifi.nullmodesleep()

Налаштовує, чи WiFi автоматично переходить у сплячий режим у NULL_MODE. Увімкнено за замовчуванням.

Примітка

Ця функція **не** зберігає свої налаштування у флеш-пам’яті, якщо автоматичний режим сну в режимі NULL_MODE небажаний, її wifi.nullmodesleep(false) потрібно викликати після ввімкнення живлення, перезапуску або виходу з глибокого сну.

Синтаксис

`wifi.nullmodesleep([enable])`

Параметри

- `enable`
- `true` Увімкнути автоматичний сон Wi-Fi у NULL_MODE. (налаштування за замовчуванням)
- `false` Вимкнути автоматичний сон Wi-Fi у NULL_MODE.

Повернення

- `sleep_enabled` Поточний/новий параметр сну NULL_MODE
  - Якщо `wifi.nullmodesleep()` викликається без аргументів, повертається поточне налаштування.
  - Якщо `wifi.nullmodesleep()` викликається з `enable` аргументом, повертається підтвердження нового налаштування.

wifi.resume()

Відновлення Wi-Fi із призупиненого стану.

Примітка

Відновлення Wi-Fi відбувається асинхронно, це означає, що запит на відновлення буде оброблено лише тоді, коли керування процесором буде передано назад до SDK (після завершення MyResumeFunction().

Зворотний виклик відновлення також виконується асинхронно та виконуватиметься лише після того, як Wi-Fi відновить нормальну роботу.

Синтаксис

`wifi.resume([resume_cb])`

Параметри

- `resume_cb` Зворотний виклик для виконання, коли WiFi виходить із режиму призупинення.
- Повернення

`nil`

приклад

```
--Resume wifi from timed or indefinite sleep
wifi.resume()

--Resume wifi from timed or indefinite sleep w/ resume callback
wifi.resume(function() print("WiFi resume") end)
```

Дивись також

- `wifi.suspend()`
- `node.sleep()`
- `node.dsleep()`

wifi.setcountry()

Встановіть інформацію про поточну країну.

Синтаксис

`wifi.setcountry(country_info)`

Параметри

- `country_info` Ця таблиця містить інформацію про країну. (Якщо в цю функцію передано порожню таблицю, буде налаштовано значення за замовчуванням.)

- country Код країни, рядок із 2 символів, що містить код країни (перелік кодів країн можна знайти тут). (За замовчуванням: "CN")
- start_ch Початковий канал (діапазон: 1-14). (За замовчуванням: 1)
- end_ch Кінцевий канал не має бути меншим за початковий (діапазон: 1-14). (За замовчуванням: 13)
- policy Параметр політики визначає, яку конфігурацію інформації про країну використовувати, інформацію про країну, надану станції точкою доступу, або локальну конфігурацію. (за замовчуванням: wifi.COUNTRY_AUTO)
  - wifi.COUNTRY_AUTO Політика країни є автоматичною, ESP8266 використовуватиме інформацію про країну, надану точкою доступу, до якої підключено станцію.
    - у режимі stationAP відповідь маяка/зонду відображатиме інформацію про країну точки доступу, до якої підключена станція.
  - wifi.COUNTRY_MANUAL Політику країни встановлено вручну, ESP8266 використовуватиме локально налаштовану інформацію про країну.

Повернення

true Якщо налаштування пройшло успішно.

приклад

```
do
  country_info={}
  country_info.country="US"
  country_info.start_ch=1
  country_info.end_ch=13
  country_info.policy=wifi.COUNTRY_AUTO;
  wifi.setcountry(country_info)
end

--compact version
```

```
  wifi.setcountry({country="US", start_ch=1, end_ch=13,
policy=wifi.COUNTRY_AUTO})

--Set defaults
  wifi.setcountry({})
```

Дивись також

`wifi.getcountry()`

wifi.setmode()

Налаштовує режим WiFi для використання. ESP8266 може працювати в одному з чотирьох режимів WiFi:

- Режим станції, де пристрій ESP8266 приєднується до існуючої мережі
- Режим точки доступу (AP), де він створює власну мережу, до якої можуть приєднатися інші
- Режим Station + AP, коли він створює власну мережу, водночас приєднуючись до іншої існуючої мережі
- WiFi вимкнено

У разі використання комбінованого режиму Station + AP той самий канал використовуватиметься для обох мереж, оскільки радіо може слухати лише один канал.

Примітка

Конфігурація Wi-Fi зберігатиметься до змін, навіть якщо пристрій вимкнено.

Синтаксис

`wifi.setmode(mode[, save])`

Параметри

- `mode` значення має бути одним із

- `wifi.STATION` коли пристрій підключено до маршрутизатора WiFi. Це часто робиться, щоб дати пристрою доступ до Інтернету.
- `wifi.SOFTAP` коли пристрій діє *лише* як точка доступу. Це дозволить вам побачити пристрій у списку мереж Wi-Fi (звичайно, якщо ви не приховаєте SSID). У цьому режимі ваш комп'ютер може підключатися до пристрою, створюючи локальну мережу. Якщо ви не зміните значення, пристрою ESP8266 буде надано локальну IP-адресу 192.168.4.1, а вашому комп’ютеру буде призначено наступну доступну IP-адресу, наприклад 192.168.4.2.
- `wifi.STATIONAP` є комбінацією `wifi.STATION` і `wifi.SOFTAP`. Це дозволяє створити локальне підключення Wi-Fi *і* підключитися до іншого роутера WiFi.
- `wifi.NULLMODE` зміна режиму WiFi на NULL_MODE переведе Wi-Fi у стан низького енергоспоживання, подібний до MODEM_SLEEP, за умови, що `wifi.nullmodesleep(false)` не було викликано.

- `save` виберіть, чи зберігати режим Wi-Fi для прошивки
  - `true` Конфігурація режиму WiFi **буде** збережена під час циклу живлення. (за умовчанням)
  - `false` Конфігурація режиму WiFi **не буде** збережена під час циклу живлення.

Повернення

поточний режим після налаштування

приклад

```
wifi.setmode(wifi.STATION)
```

Дивись також

`wifi.getmode()` `wifi.getdefaultmode()`

wifi.setphymode()

Встановлює фізичний режим WiFi.

- `wifi.PHYMODE_B` 802.11b, більший діапазон, низька швидкість передачі, більше споживання струму
- `wifi.PHYMODE_G` 802.11g, середній радіус дії, середня швидкість передачі даних, середнє споживання струму
- `wifi.PHYMODE_N` 802.11n, найменший радіус дії, висока швидкість передачі, найменше споживання струму (ЛИШЕ ДЛЯ СТАНЦІЇ) Інформація з таблиці даних Espressif v4.3

Параметри	Типове споживання електроенергії
Tx 802.11b, CCK 11 Мбіт/с, P OUT=+17 дБм	170 мА
Tx 802.11g, OFDM 54 Мбіт/с, P OUT =+15 дБм	140 мА
Tx 802.11n, MCS7 65 Мбіт/с, P OUT =+13 дБм	120 мА
Rx 802.11b, довжина пакету 1024 байти, -80 дБм	50 мА
Rx 802.11g, довжина пакету 1024 байти, -70 дБм	56 мА
Rx 802.11n, довжина пакета 1024 байти, -65 дБм	56 мА

Синтаксис

`wifi.setphymode(mode)`

Параметри

`mode` одне з наступного

- `wifi.PHYMODE_B`

- `wifi.PHYMODE_G`
- `wifi.PHYMODE_N`

Повернення

фізичний режим після налаштування

Дивись також

`wifi.getphymode()`

wifi.setmaxtxpower()

Встановлює максимальну потужність передачі WiFi. Це налаштування не зберігається протягом циклів живлення, а в документації Espressif SDK не вказано, чи зберігається налаштування після глибокого сну. Значення за замовчуванням, яке використовується, зчитується з байта 34 ініціальних даних ESP8266, і, отже, його значення визначається виробником.

Значення за замовчуванням, 82, відповідає максимальній потужності передачі. Зменшення цього параметра може зменшити споживання електроенергії пристроями, що працюють від батареї.

Синтаксис

`wifi.setmaxtxpower(max_tpw)`

Параметри

`max_tpw` максимальне значення RF Tx Power, одиниці: 0,25 дБм, діапазон [0, 82].

Повернення

`nil`

Дивись також

`flash SDK init data`

wifi.startsmart()

Починає автоматичне налаштування, у разі успіху автоматично встановлює SSID і пароль.

Призначено для використання з програмами SmartConfig, такими як програма Espressif для Android та iOS.

Можна використовувати лише в `wifi.STATION` режимі.

Синтаксис

`wifi.startsmart(type, callback)`

Параметри

- `type` 0 для ESP_TOUCH або 1 для AIR_KISS.
- `callback` функція зворотного виклику форми `function(ssid, password) end`, яка викликається після налаштування.

Повернення

`nil`

приклад

```
wifi.setmode(wifi.STATION)
wifi.startsmart(0,
    function(ssid, password)
        print(string.format("Success. SSID:%s ; PASSWORD:%s", ssid,
password))
    end
)
```

Дивись також

`wifi.stopsmart()`

wifi.stopsmart()

Зупиняє процес розумного налаштування.

Синтаксис

`wifi.stopsmart()`

Параметри

немає

Повернення

`nil`

Дивись також

`wifi.startsmart()`

wifi.suspend()

Призупинення Wi-Fi, щоб зменшити поточне споживання.

Примітка

Призупинення Wi-Fi відбувається асинхронно, це означає, що запит на призупинення буде оброблено лише тоді, коли керування процесором буде передано назад до SDK (після завершення MySuspendFunction(). Зворотний виклик призупинення також виконується асинхронно й виконуватиметься лише після успішного призупинення Wi-Fi.

Синтаксис

`wifi.suspend({duration[, suspend_cb, resume_cb, preserve_mode]})`

Параметри

- `duration` Тривалість призупинення в мікросекундах (мкс). Якщо вказано тривалість призупинення `0`, призупинення буде невизначеним (діапазон: 0 або 50000–268435454 мкс (0:4:28.000454))
- `suspend_cb` Зворотний виклик для виконання, коли Wi-Fi призупинено. (необов'язково)
- `resume_cb` Зворотний виклик для виконання, коли WiFi виходить із режиму призупинення. (необов'язково)

- preserve_mode зберегти поточний режим WiFi через сплячий режим вузла. (Необов'язково, за замовчуванням: true)
- Якщо значення true, режими Station і StationAP автоматично відновлять підключення до попередньо налаштованої точки доступу, коли ESP8266 відновить роботу.
- Якщо false, відмініть режим WiFi і залиште ESP8266 у wifi.NULL_MODE. Режим Wi-Fi буде відновлено до вихідного режиму після перезавантаження.

Повернення

- suspend_state якщо параметри не надано, буде повернуто поточний стан призупинення WiFi держави:
- 0 Wi-Fi не активний.
- 1 Призупинення Wi-Fi очікує на розгляд. (Очікування неактивного завдання)
- 2 WiFi призупинено.

приклад

```
--get current wifi suspension state
print(wifi.suspend())

--Suspend WiFi for 10 seconds with suspend/resume callbacks
 cfg={}
 cfg.duration=10*1000*1000
 cfg.resume_cb=function() print("WiFi resume") end
 cfg.suspend_cb=function() print("WiFi suspended") end

 wifi.suspend(cfg)

--Suspend WiFi for 10 seconds with suspend/resume callbacks and discard
WiFi mode
 cfg={}
 cfg.duration=10*1000*1000
 cfg.resume_cb=function() print("WiFi resume") end
 cfg.suspend_cb=function() print("WiFfi suspended") end
 cfg.preserve_mode=false
```

```
wifi.suspend(cfg)
```

Дивись також

- wifi.resume()
- node.sleep()
- node.dsleep()

МОДУЛЬ WIFI.STA

wifi.sta.autoconnect()

Автоматичне підключення до точки доступу в режимі станції.

Синтаксис

`wifi.sta.autoconnect(auto)`

Параметри

`auto` 0, щоб вимкнути автоматичне підключення, 1, щоб увімкнути автоматичне підключення

Повернення

`nil`

приклад

```
wifi.sta.autoconnect(1)
```

Дивись також

- `wifi.sta.config()`
- `wifi.sta.connect()`
- `wifi.sta.disconnect()`

wifi.sta.changeap()

Виберіть точку доступу зі списку, наданого користувачем `wifi.sta.getapinfo()`

Синтаксис

`wifi.sta.changeap(ap_index)`

Параметри

ap_index Індекс точки доступу, на яку ви хочете змінитися. (Діапазон: 1-5) - відповідає індексу, який використовується wifi.sta.getapinfo() і wifi.sta.getapindex()

Повернення

- true Успіх
- false

приклад

```
wifi.sta.changeap(4)
```

Дивись також

- wifi.sta.getapinfo()
- wifi.sta.getapindex()

wifi.sta.clearconfig()

Очищає поточну збережену конфігурацію станції WiFi, видаляючи її з флеш-пам'яті. Може бути корисним для певних сценаріїв скидання до заводських налаштувань, коли повне скидання node.restore()не потрібне, або для підготовки до використання налаштування кінцевого користувача, щоб SoftAP міг зафіксуватися на одному апаратному радіоканалі.

Синтаксис

wifi.sta.clearconfig()

Параметри

немає

Повернення

- true Успіх

- `false`

Дивись також

- `wifi.sta.config()`
- `node.restore()`

wifi.sta.config()

Встановлює конфігурацію станції WiFi.

Примітка

Під час запуску ініціалізації не рекомендується вважати, що Wi-Fi підключено в будь-який час. Статус підключення Wi-Fi слід перевіряти або за допомогою зворотного виклику події WiFi, або шляхом опитування статусу на таймері.

Синтаксис

`wifi.sta.config(station_config)`

Параметри

- `station_config` таблиця, що містить конфігураційні дані станції
  - `ssid` рядок менше 32 байтів.
  - `pwd` рядок 0-64. Порожній рядок означає відкриту точку доступу WiFi. *Примітка: WPA вимагає мінімум 8 символів, але ESP8266 також може підключатися до точки доступу WEP (40-бітний ключ WEP можна надати як відповідний 5-символьний рядок ASCII).*
  - `auto` за замовчуванням значення true
    - `true` щоб увімкнути автоматичне підключення та підключитися до точки доступу, отже, `auto=true` немає необхідності викликати `wifi.sta.connect()`

- `false` щоб вимкнути автоматичне підключення та залишитися відключеним від точки доступу

- `bssid` рядок, що містить MAC-адресу точки доступу (необов’язково)
  - Ви можете встановити BSSID, якщо у вас є кілька точок доступу з однаковим SSID.
  - Якщо ви встановлюєте BSSID для певного SSID і бажаєте налаштувати станцію для підключення лише до того самого SSID без вимоги BSSID, ви ПОВИННІ спочатку налаштувати станцію для іншого SSID, а потім підключитися до потрібного SSID
  - Допустимі наступні формати:
    - "DE:C1:A5:51:F1:ED"
    - "AC-1D-1C-B1-0B-22"
    - "DE AD BE EF 7A C0"
- `save` Збережіть конфігурацію станції для флеш-пам’яті.
  - `true` конфігурація **буде** збережена протягом циклу живлення. (За замовчуванням).
  - `false` конфігурація **не буде** збережена під час циклу живлення.
- Зворотні виклики подій будуть доступні, лише якщо `WIFI_SDK_EVENT_MONITOR_ENABLE` не закоментовано `user_config.h`
  - Будь ласка, зверніть увагу: щоб забезпечити обробку всіх подій станції під час завантаження, усі відповідні зворотні виклики мають бути зареєстровані якомога раніше за `init.lua` допомогою `wifi.sta.config()` або `wifi.eventmon.register()`.
  - `connected_cb`: Зворотний виклик для виконання, коли станція підключена до точки доступу. (необов'язково)
    - Елементи, повернуті в таблицю:

        - `SSID` : SSID точки доступу. (формат: рядок)
        - `BSSID` : BSSID точки доступу. (формат: рядок)
        - `channel` : канал, на якому працює точка доступу. (формат: число)
- `disconnected_cb` : Зворотний виклик для виконання, коли станція відключена від точки доступу. (необов'язково)
    - Елементи, повернуті в таблицю:
        - `SSID` : SSID точки доступу. (формат: рядок)
        - `BSSID` : BSSID точки доступу. (формат: рядок)
        - `reason` : Див. wifi.eventmon.reason нижче. (формат: число)
- `authmode_change_cb` : Зворотний виклик для виконання, коли точка доступу змінила режим авторизації. (необов'язково)
    - Елементи, повернуті в таблицю:
    - `old_auth_mode` : Старий режим авторизації Wi-Fi. (формат: число)
    - `new_auth_mode` : Новий режим авторизації Wi-Fi. (формат: число)
- `got_ip_cb` : Зворотний виклик для виконання, коли станція отримала IP-адресу від точки доступу. (необов'язково)
    - Елементи, повернуті в таблицю:
        - `IP` : IP-адреса, призначена станції. (формат: рядок)
        - `netmask` : Маска підмережі. (формат: рядок)
        - `gateway` : IP-адреса точки доступу, до якої підключена станція. (формат: рядок)
- `dhcp_timeout_cb` : Запит станції DHCP минув. (необов'язково)

- Повернено порожню таблицю.

Повернення

- `true` Успіх
- `false`

приклад

```
--connect to Access Point (DO NOT save config to flash)
station_cfg={}
station_cfg.ssid="NODE-AABBCC"
station_cfg.pwd="password"
station_cfg.save=false
wifi.sta.config(station_cfg)

--connect to Access Point (DO save config to flash)
station_cfg={}
station_cfg.ssid="NODE-AABBCC"
station_cfg.pwd="password"
station_cfg.save=true
wifi.sta.config(station_cfg)

--connect to Access Point with specific MAC address (DO save config to
flash)
station_cfg={}
station_cfg.ssid="NODE-AABBCC"
station_cfg.pwd="password"
station_cfg.bssid="AA:BB:CC:DD:EE:FF"
wifi.sta.config(station_cfg)

--configure station but don't connect to Access point (DO save config to
flash)
station_cfg={}
station_cfg.ssid="NODE-AABBCC"
station_cfg.pwd="password"
station_cfg.auto=false
wifi.sta.config(station_cfg)
```

Дивись також

- `wifi.sta.clearconfig()`
- `wifi.sta.connect()`
- `wifi.sta.disconnect()`
- `wifi.sta.getapinfo()`

wifi.sta.connect()

Підключається до налаштованої точки доступу в режимі станції.

Синтаксис

`wifi.sta.connect([connected_cb])`

Параметри

- `connected_cb` : Зворотний виклик для виконання, коли станція підключена до точки доступу. (необов'язково)
  - Елементи, повернуті в таблицю:
    - `SSID` : SSID точки доступу. (формат: рядок)
    - `BSSID` : BSSID точки доступу. (формат: рядок)
    - `channel` : канал, на якому працює точка доступу. (формат: число)

Повернення

`nil`

Дивись також

- `wifi.sta.disconnect()`
- `wifi.sta.config()`

wifi.sta.disconnect()

Відключається від точки доступу в режимі станції.

Синтаксис

`wifi.sta.disconnect([disconnected_cb])`

Параметри

- `disconnected_cb` : Зворотний виклик для виконання, коли станція відключена від точки доступу. (необов'язково)
  - Елементи, повернуті в таблицю:
    - `SSID` : SSID точки доступу. (формат: рядок)

- BSSID : BSSID точки доступу. (формат: рядок)
- reason : Див. wifi.eventmon.reason нижче. (формат: число)

Повернення

nil

Дивись також

- wifi.sta.config()
- wifi.sta.connect()

wifi.sta.getap()

Сканує список точок доступу як таблицю Lua у функцію зворотного виклику.

Синтаксис

wifi.sta.getap([[cfg], format,] callback(table))

Параметри

- cfg таблиця, яка містить конфігурацію сканування
  - ssid SSID == нуль, не фільтрувати SSID
  - bssid BSSID == нуль, не фільтрувати BSSID
  - channel канал == 0, сканувати всі канали, інакше сканувати встановлений канал (за замовчуванням 0)
  - show_hidden show_hidden == 1, отримати інформацію для маршрутизатора з прихованим SSID (за замовчуванням 0)
- format виберіть формат вихідної таблиці, за замовчуванням 0
  - 0: старий формат (SSID: Authmode, RSSI, BSSID, Channel), будь-які повторювані SSID буде відкинуто
  - 1: новий формат (BSSID: SSID, RSSI, режим авторизації, канал)

- `callback(table)` функція зворотного виклику для отримання таблиці AP після завершення сканування. Ця функція отримує таблицю, ключем є BSSID, значенням є інша інформація у форматі: SSID, RSSID, режим авторизації, канал.

Повернення

`nil`

приклад

```
-- print AP list in old format (format not defined)
function listap(t)
    for k,v in pairs(t) do
        print(k.." : "..v)
    end
end
wifi.sta.getap(listap)

-- Print AP list that is easier to read
function listap(t) -- (SSID : Authmode, RSSI, BSSID, Channel)
    print("\n"..string.format("%32s","SSID").."\tBSSID\t\t\t\t
RSSI\t\tAUTHMODE\tCHANNEL")
    for ssid,v in pairs(t) do
        local authmode, rssi, bssid, channel = string.match(v,
"([^,]+),([^,]+),([^,]+),([^,]+)")
        print(string.format("%32s",ssid).."\t"..bssid.."\t
"..rssi.."\t\t"..authmode.."\t\t\t"..channel)
    end
end
wifi.sta.getap(listap)

-- print AP list in new format
function listap(t)
    for k,v in pairs(t) do
        print(k.." : "..v)
    end
end
wifi.sta.getap(1, listap)

-- Print AP list that is easier to read
function listap(t) -- (SSID : Authmode, RSSI, BSSID, Channel)
    print("\n\t\t\tSSID\t\t\t\t\tBSSID\t\t\t
RSSI\t\tAUTHMODE\t\tCHANNEL")
    for bssid,v in pairs(t) do
        local ssid, rssi, authmode, channel = string.match(v,
"([^,]+),([^,]+),([^,]+),([^,]*)")
```

```
        print(string.format("%32s",ssid).."\t"..bssid.."\t
"..rssi.."\t\t"..authmode.."\t\t\t"..channel)
    end
end
wifi.sta.getap(1, listap)

--check for specific AP
function listap(t)
    print("\n\t\t\tSSID\t\t\t\t\tBSSID\t\t\t
RSSI\t\tAUTHMODE\t\tCHANNEL")
    for bssid,v in pairs(t) do
        local ssid, rssi, authmode, channel = string.match(v,
"([^,]+),([^,]+),([^,]+),([^,]*)")
        print(string.format("%32s",ssid).."\t"..bssid.."\t
"..rssi.."\t\t"..authmode.."\t\t\t"..channel)
    end
end
scan_cfg = {}
scan_cfg.ssid = "myssid"
scan_cfg.bssid = "AA:AA:AA:AA:AA:AA"
scan_cfg.channel = 0
scan_cfg.show_hidden = 1
wifi.sta.getap(scan_cfg, 1, listap)

--get RSSI for currently configured AP
function listap(t)
    for bssid,v in pairs(t) do
        local ssid, rssi, authmode, channel = string.match(v,
"([^,]+),([^,]+),([^,]+),([^,]*)")
        print("CURRENT RSSI IS: "..rssi)
    end
end
ssid, tmp, bssid_set, bssid=wifi.sta.getconfig()

scan_cfg = {}
scan_cfg.ssid = ssid
if bssid_set == 1 then scan_cfg.bssid = bssid else scan_cfg.bssid = nil
end
scan_cfg.channel = wifi.getchannel()
scan_cfg.show_hidden = 0
ssid, tmp, bssid_set, bssid=nil, nil, nil, nil
wifi.sta.getap(scan_cfg, 1, listap)
```

Дивись також

wifi.sta.getip()

wifi.sta.getapindex()

Отримати індекс поточної точки доступу, що зберігається в кеші AP.

Синтаксис

`wifi.sta.getapindex()`

Параметри

немає

Повернення

`current_index` індекс поточної вибраної точки доступу. (Діапазон: 1-5)

приклад

```
print("the index of the currently selected AP is: "..wifi.sta.getapindex())
```

Дивись також

- `wifi.sta.getapindex()`
- `wifi.sta.getapinfo()`
- `wifi.sta.changeap()`

wifi.sta.getapinfo()

Отримує інформацію про точки доступу, кешовані станцією ESP8266.

Примітка

Будь-які точки доступу, налаштовані з вимкненим збереженням, wifi.sta.config({save=false})заповнюватимуть цей список (перезаписуючи точки доступу, збережені у флеш-пам’яті), до перезавантаження.

Синтаксис

`wifi.sta.getapinfo()`

Параметри

`nil`

Повернення

- `ap_info`
  - `qty` кількість повернених AP
  - `1-5` індекс АП. (індекс відповідає індексу, який використовує `wifi.sta.changeap()` та `wifi.sta.getapindex()`)
  - `ssid` ssid точки доступу
  - `pwd` пароль для точки доступу, `nil` якщо пароль не було налаштовано
  - `bssid` MAC-адреса точки доступу
  - `nil` буде повернено, якщо MAC-адреса не була налаштована під час налаштування станції.

приклад

```
--print stored access point info
do
  for k,v in pairs(wifi.sta.getapinfo()) do
    if (type(v)=="table") then
      print(" "..k.." : "..type(v))
      for k,v in pairs(v) do
        print("\t\t"..k.." : "..v)
      end
    else
      print(" "..k.." : "..v)
    end
  end
end

--print stored access point info(formatted)
do
  local x=wifi.sta.getapinfo()
  local y=wifi.sta.getapindex()
  print("\n Number of APs stored in flash:", x.qty)
  print(string.format("  %-6s %-32s %-64s %-18s", "index:", "SSID:",
"Password:", "BSSID:"))
  for i=1, (x.qty), 1 do
    print(string.format(" %s%-6d %-32s %-64s %-18s",(i==y and ">" or " "),
i, x[i].ssid, x[i].pwd and x[i].pwd or type(nil), x[i].bssid and
x[i].bssid or type(nil)))
```

```
  end
end
```

Дивись також

- wifi.sta.getapindex()
- wifi.sta.setaplimit()
- wifi.sta.changeap()
- wifi.sta.config()

wifi.sta.getbroadcast()

Отримує широкомовну адресу в станційному режимі.

Синтаксис

wifi.sta.getbroadcast()

Параметри

nil

Повернення

широкомовна адреса як рядок, наприклад "192.168.0.255", повертає, nil якщо IP-адреса = "0.0.0.0".

Дивись також

wifi.sta.getip()

wifi.sta.getconfig()

Отримує конфігурацію станції WiFi.

Синтаксис

wifi.sta.getconfig()

Параметри

- return_table

- `true` повертає дані в таблиці
- `false` повертає дані в старому форматі (за замовчуванням)

Повернення

Якщо `return_table` є `true`:

- `config_table`
  - `ssid` ssid точки доступу.
  - `pwd` пароль до точки доступу, `nil` якщо пароль не було налаштовано
  - `bssid_set` повернеться, `true` якщо станцію було налаштовано спеціально для підключення до точки доступу з відповідним `bssid`.
  - `bssid` Якщо підключення до налаштованої точки доступу встановлено, це поле міститиме MAC-адресу точки доступу. Інакше буде повернено "ff:ff:ff:ff:ff:ff".

Якщо `return_table` є `false`:

- ssid, пароль, bssid_set, bssid, якщо `bssid_set` дорівнює, `0` то `bssid` не має значення

приклад

```
--Get current Station configuration (NEW FORMAT)
do
local sta_config=wifi.sta.getconfig(true)
print(string.format("\tCurrent station
config\n\tssid:\"%s\"\tpassword:\"%s\"\n\tbssid:\"%s\"\tbssid_set:%s",
sta_config.ssid, sta_config.pwd, sta_config.bssid, (sta_config.bssid_set
and "true" or "false")))
end

--Get current Station configuration (OLD FORMAT)
ssid, password, bssid_set, bssid=wifi.sta.getconfig()
print("\nCurrent Station configuration:\nSSID : "..ssid
.."\nPassword  : "..password
.."\nBSSID_set  : "..bssid_set
.."\nBSSID: "..bssid.."\n")
ssid, password, bssid_set, bssid=nil, nil, nil, nil
```

Дивись також

- `wifi.sta.getdefaultconfig()`
- `wifi.sta.connect()`
- `wifi.sta.disconnect()`

wifi.sta.getdefaultconfig()

Отримує стандартну конфігурацію станції Wi-Fi, збережену у флеш-пам'яті.

Синтаксис

`wifi.sta.getdefaultconfig(return_table)`

Параметри

- `return_table`
  - `true` повертає дані в таблиці
  - `false` повертає дані в старому форматі (за замовчуванням)

Повернення

Якщо `return_table` є `true`:

- `config_table`
  - `ssid` ssid точки доступу.
  - `pwd` пароль до точки доступу, `nil` якщо пароль не було налаштовано
  - `bssid_set` повернеться, `true` якщо станцію було налаштовано спеціально для підключення до точки доступу з відповідним `bssid`.
  - `bssid` Якщо підключення до налаштованої точки доступу встановлено, це поле міститиме MAC-адресу точки доступу. Інакше буде повернено "ff:ff:ff:ff:ff:ff".

Якщо `return_table` є `false`:

- ssid, пароль, bssid_set, bssid, якщо `bssid_set` дорівнює, `0` то `bssid` не має значення

приклад

```
--Get default Station configuration (NEW FORMAT)
do
local def_sta_config=wifi.sta.getdefaultconfig(true)
print(string.format("\tDefault station
config\n\tssid:\"%s\"\tpassword:\"%s\"\n\tbssid:\"%s\"\tbssid_set:%s",
def_sta_config.ssid, def_sta_config.pwd, def_sta_config.bssid,
(def_sta_config.bssid_set and "true" or "false")))
end

--Get default Station configuration (OLD FORMAT)
ssid, password, bssid_set, bssid=wifi.sta.getdefaultconfig()
print("\nCurrent Station configuration:\nSSID : "..ssid
.."\nPassword  : "..password
.."\nBSSID_set  : "..bssid_set
.."\nBSSID: "..bssid.."\n")
ssid, password, bssid_set, bssid=nil, nil, nil, nil
```

Дивись також

- `wifi.sta.getconfig()`
- `wifi.sta.connect()`
- `wifi.sta.disconnect()`

wifi.sta.gethostname()

Отримує назву хоста поточної станції.

Синтаксис

`wifi.sta.gethostname()`

Параметри

немає

Повернення

поточне налаштоване ім'я хоста

приклад

```
print("Current hostname is: \""..wifi.sta.gethostname().."\"")
```

wifi.sta.getip()

Отримує IP-адресу, маску мережі та адресу шлюзу в режимі станції.

Синтаксис

`wifi.sta.getip()`

Параметри

немає

Повернення

IP-адреса, маска мережі, адреса шлюзу у вигляді рядка, наприклад «192.168.0.111». Повертає `nil`, якщо IP = "0.0.0.0".

приклад

```
-- print current IP address, netmask, gateway
print(wifi.sta.getip())
-- 192.168.0.111  255.255.255.0  192.168.0.1
ip = wifi.sta.getip()
print(ip)
-- 192.168.0.111
ip, nm = wifi.sta.getip()
print(nm)
-- 255.255.255.0
```

Дивись також

`wifi.sta.getmac()`

wifi.sta.getmac()

Отримує MAC-адресу в режимі станції.

Синтаксис

`wifi.sta.getmac()`

Параметри

немає

Повернення

MAC-адреса як рядок, наприклад "18:fe:34:a2:d7:34"

Дивись також

wifi.sta.getip()

wifi.sta.getrssi()

Отримує RSSI (індикатор потужності отриманого сигналу) точки доступу, до якої підключилася станція ESP8266.

Синтаксис

wifi.sta.getrssi()

Параметри

немає

Повернення

- Якщо станцію підключено до точки доступу, rssi повертається.
- Якщо станція не підключена до точки доступу, nil повертається.

приклад

```
RSSI=wifi.sta.getrssi()

print("RSSI is", RSSI)
```

wifi.sta.setaplimit()

Встановлює максимальну кількість точок доступу для зберігання у флеш-пам’яті. - Це значення записане для спалаху

Примітка

Якщо збережено 5 точок доступу, а обмеження AP встановлено на 4, AP з індексом 5 залишатиметься до node.restore() виклику або обмеження AP встановлено на 5, і AP буде перезаписано.

Синтаксис

wifi.sta.setaplimit(qty)

Параметри

qty Кількість точок доступу для зберігання у флеш-пам’яті. Діапазон: 1-5 (за замовчуванням: 1)

Повернення

- true Успіх
- false

приклад

```
wifi.sta.setaplimit(5)
```

Дивись також

- wifi.sta.getapinfo()

wifi.sta.sethostname()

Встановлює назву хоста станції.

Синтаксис

wifi.sta.sethostname(hostname)

Параметри

hostname має містити лише літери, цифри та дефіси ('-') і мати 32 символи або менше, причому перший і останній символи є буквено-цифровими

Повернення

- true Успіх
- false

приклад

```
if (wifi.sta.sethostname("ESP8266") == true) then
    print("hostname was successfully changed")
else
    print("hostname was not changed")
end
```

wifi.sta.setip()

Встановлює IP-адресу, маску мережі, адресу шлюзу в режимі станції.

Синтаксис

`wifi.sta.setip(cfg)`

Параметри

`cfg` Таблиця містить IP-адресу, маску мережі та шлюз

```
{
  ip = "192.168.0.111",
  netmask = "255.255.255.0",
  gateway = "192.168.0.1"
}
```

Повернення

істина, якщо успіх, хибність в іншому випадку

Дивись також

`wifi.sta.setmac()`

wifi.sta.setmac()

Встановлює MAC-адресу в станційному режимі.

Синтаксис

`wifi.sta.setmac(mac)`

Параметри

MAC-адреса в рядку, наприклад "DE:AD:BE:EF:7A:C0"

Повернення

істина, якщо успіх, хибність в іншому випадку

приклад

```
print(wifi.sta.setmac("DE:AD:BE:EF:7A:C0"))
```

Дивись також

`wifi.sta.setip()`

wifi.sta.sleeptype()

Налаштовує тип сну WiFi-модему, який буде використовуватися, коли станція підключена до точки доступу.

Синтаксис

`wifi.sta.sleeptype(type_wanted)`

Параметри

`type_wanted` одне з наступного:

- `wifi.NONE_SLEEP` щоб модем був увімкненим постійно
- `wifi.LIGHT_SLEEP` дозволити ЦП вимкнутися за певних обставин
- `wifi.MODEM_SLEEP` щоб максимально вимкнути модем

Повернення

Фактичний режим сну встановлено як один із `wifi.NONE_SLEEP`, `wifi.LIGHT_SLEEP` або `wifi.MODEM_SLEEP`.

wifi.sta.status()

Отримує поточний статус у станційному режимі.

Синтаксис

`wifi.sta.status()`

Параметри

`nil`

Повернення

Поточний стан, який може бути одним із таких:

- `wifi.STA_IDLE`
- `wifi.STA_CONNECTING`
- `wifi.STA_WRONGPWD`

МОДУЛЬ WIFI.AP

wifi.ap.config()

Встановлює SSID і пароль у режимі AP.

Синтаксис

`wifi.ap.config(cfg)`

Параметри

- `cfg` таблиця для зберігання конфігурації
  - `ssid` SSID символи 1-32
  - `pwd` символи пароля 8-64
  - `auth` метод автентифікації, один із `wifi.OPEN` (за амовчуванням), `wifi.WPA_PSK`, `wifi.WPA2_PSK`, `wifi.WPA_WPA2_PSK`
  - `channel` номер каналу 1-14 за замовчуванням = 6
  - `hidden` false = не приховано, true = приховано, default = false
  - `max` максимальна кількість підключень 1-4 за замовчуванням=4
  - `beacon` інтервал часу маяка в діапазоні 100-60000, за замовчуванням = 100
  - `save` зберегти конфігурацію для прошивки.
    - `true` конфігурація **буде** збережена протягом циклу живлення. (за умовчанням)
    - `false` конфігурація **не буде** збережена під час циклу живлення.
  - Зворотні виклики подій будуть доступні, лише якщо `WIFI_SDK_EVENT_MONITOR_ENABLE` не закоментовано

- staconnected_cb: Зворотний виклик виконується, коли новий клієнт підключається до точки доступу. (необов'язково)
  - Елементи, повернуті в таблицю:
    - MAC: MAC-адреса клієнта, який підключився.
    - AID: SDK не надає подробиць щодо цього значення, що повертається.
- stadisconnected_cb: Зворотний виклик виконується, коли клієнт від’єднується від точки доступу. (необов'язково)
  - Елементи, повернуті в таблицю:
    - MAC: MAC-адреса клієнта, який відключився.
    - AID: SDK не надає подробиць щодо цього значення, що повертається.
- probereq_cb: Зворотний виклик виконується, коли отримано запит на зондування. (необов'язково)
  - Елементи, повернуті в таблицю:
    - MAC: MAC-адреса клієнта, який перевіряє точку доступу.
    - RSSI: Індикатор рівня отриманого сигналу клієнта.

Повернення

- true Успіх
- false

приклад:

```
cfg={}
cfg.ssid="myssid"
cfg.pwd="mypassword"
wifi.ap.config(cfg)
```

wifi.ap.deauth()

Виключає (примусово видаляє) клієнта з точки доступу ESP, надсилаючи відповідний пакет керування IEEE802.11 (спочатку) і видаляючи клієнта з його структур даних (після).

Використовується код причини IEEE802.11 2 для «Попередня автентифікація більше недійсна» (AUTH_EXPIRE).

Синтаксис

`wifi.ap.deauth([MAC])`

Параметри

- `MAC` адреса станції, яку необхідно вимкнути.
  - Примітка: якщо це поле залишити незаповненим, усі підключені станції буде скасовано.

Повернення

Повертає true, якщо не викликається, коли ESP перебуває в режимі STATION

приклад

```
allowed_mac_list={"18:fe:34:00:00:00", "18:fe:34:00:00:01"}

wifi.eventmon.register(wifi.eventmon.AP_STACONNECTED, function(T)
  print("\n\tAP - STATION CONNECTED".."\n\tMAC: "..T.MAC.."\n\tAID: 
"..T.AID)
  if(allowed_mac_list~=nil) then
    for _, v in pairs(allowed_mac_list) do
      if(v == T.MAC) then return end
    end
  end
  wifi.ap.deauth(T.MAC)
  print("\tStation DeAuthed!")
end)
```

Дивись також

`wifi.eventmon.register()` `wifi.eventmon.reason()`

wifi.ap.getbroadcast()

Отримує широкомовну адресу в режимі AP.

Синтаксис

`wifi.ap.getbroadcast()`

Параметри

немає

Повернення

широкомовна адреса в рядку, наприклад "192.168.0.255", повертає, `nil` якщо IP-адреса = "0.0.0.0".

приклад

```
bc = wifi.ap.getbroadcast()
print(bc)
-- 192.168.0.255
```

Дивись також

`wifi.ap.getip()`

wifi.ap.getclient()

Отримує таблицю клієнтів, підключених до пристрою в режимі точки доступу.

Синтаксис

`wifi.ap.getclient()`

Параметри

немає

Повернення

таблиця підключених клієнтів

приклад

```
table={}
table=wifi.ap.getclient()
for mac,ip in pairs(table) do
    print(mac,ip)
end
```

```
-- or shorter
for mac,ip in pairs(wifi.ap.getclient()) do
    print(mac,ip)
end
```

wifi.ap.getconfig()

Отримує поточну конфігурацію SoftAP.

Синтаксис

`wifi.ap.getconfig(return_table)`

Параметри

- `return_table`
  - `true` повертає дані в таблиці
  - `false` повертає дані в старому форматі (за замовчуванням)

Повернення

Якщо `return_table` вірно:

- `config_table`
  - `ssid` Назва мережі
  - `pwd` Пароль, якщо `nil` пароль не налаштовано - `auth` метод автентифікації (`wifi.OPEN`, `wifi.WPA_PSK` або) `wifi.WPA2_PSKwifi.WPA_WPA2_PSK`
  - `channel` Номер каналу
  - `hidden` `false` = не прихований, `true` = прихований
  - `max` Максимальна кількість клієнтських підключень
  - `beacon` Маяковий інтервал

Якщо `return_table` false:

- ssid, пароль, якщо `bssid_set` дорівнює 0, то `bssid` не має значення

приклад

```
--Get SoftAP configuration table (NEW FORMAT)
do
  print("\n  Current SoftAP configuration:")
  for k,v in pairs(wifi.ap.getconfig(true)) do
      print("   "..k.." :",v)
  end
end

--Get current SoftAP configuration (OLD FORMAT)
do
  local ssid, password=wifi.ap.getconfig()
  print("\n  Current SoftAP configuration:\n   SSID : "..ssid..
    "\n   Password  :",password)
  ssid, password=nil, nil
end
```

wifi.ap.getdefaultconfig()

Отримує стандартну конфігурацію SoftAP, збережену у флеш-пам'яті.

Синтаксис

`wifi.ap.getdefaultconfig(return_table)`

Параметри

- `return_table`
  - `true` повертає дані в таблиці
  - `false` повертає дані в старому форматі (за замовчуванням)

Повернення

Якщо `return_table` вірно:

- `config_table`
  - `ssid` Назва мережі

- `pwd` Пароль, якщо `nil` пароль не налаштовано - `auth` метод автентифікації (`wifi.OPEN`, `wifi.WPA_PSK` або) `wifi.WPA2_PSKwifi.WPA_WPA2_PSK`
- `channel` Номер каналу
- `hidden` `false` = не прихований, `true` = прихований
- `max` Максимальна кількість клієнтських підключень
- `beacon` Маяковий інтервал

Якщо `return_table` false:

- ssid, пароль, якщо `bssid_set` дорівнює 0, то `bssid` не має значення

приклад

```
--Get default SoftAP configuration table (NEW FORMAT)
do
  print("\n  Default SoftAP configuration:")
  for k,v in pairs(wifi.ap.getdefaultconfig(true)) do
      print("   "..k.." :",v)
  end
end

--Get default SoftAP configuration (OLD FORMAT)
do
  local ssid, password=wifi.ap.getdefaultconfig()
  print("\n  Default SoftAP configuration:\n   SSID : "..ssid..
    "\n   Password  :",password)
  ssid, password=nil, nil
end
```

wifi.ap.getip()

Отримує IP-адресу, маску мережі та шлюз у режимі AP.

Синтаксис

`wifi.ap.getip()`

Параметри

немає

Повернення

IP-адреса, маска мережі, адреса шлюзу у вигляді рядка, наприклад "192.168.0.111", повертається, `nil` якщо IP-адреса = "0.0.0.0".

приклад

```
-- print current ip, netmask, gateway
print(wifi.ap.getip())
-- 192.168.4.1  255.255.255.0  192.168.4.1
ip = wifi.ap.getip()
print(ip)
-- 192.168.4.1
ip, nm = wifi.ap.getip()
print(nm)
-- 255.255.255.0
ip, nm, gw = wifi.ap.getip()
print(gw)
-- 192.168.4.1
```

Дивись також

- `wifi.ap.getmac()`

wifi.ap.getmac()

Отримує MAC-адресу в режимі AP.

Синтаксис

`wifi.ap.getmac()`

Параметри

немає

Повернення

MAC-адреса у вигляді рядка, наприклад "1A-33-44-FE-55-BB"

Дивись також

`wifi.ap.getip()`

wifi.ap.setip()

Встановлює IP-адресу, маску мережі та адресу шлюзу в режимі AP.

Синтаксис

wifi.ap.setip(cfg)

Параметри

cfg Таблиця містить IP-адресу, маску мережі та шлюз

Повернення

істина, якщо успішно, хибна в іншому випадку

приклад

```
cfg =
{
    ip="192.168.1.1",
    netmask="255.255.255.0",
    gateway="192.168.1.1"
}
wifi.ap.setip(cfg)
```

Дивись також

wifi.ap.setmac()

wifi.ap.setmac()

Встановлює MAC-адресу в режимі AP.

Синтаксис

wifi.ap.setmac(mac)

Параметри

MAC-адреса в рядку байтів, наприклад "AC-1D-1C-B1-0B-22"

Повернення

істина, якщо успіх, хибність в іншому випадку

приклад

```
print(wifi.ap.setmac("AC-1D-1C-B1-0B-22"))
```

Дивись також

wifi.ap.setip()

МОДУЛЬ WIFI.AP.DHCP

wifi.ap.dhcp.config()

Налаштовує службу dhcp. Наразі підтримується лише встановлення початкової адреси пулу адрес dhcp.

Синтаксис

`wifi.ap.dhcp.config(dhcp_config)`

Параметри

`dhcp_config` таблиця, що містить початковий IP пулу адрес DHCP, напр. "192.168.1.100"

Повернення

`pool_startip`, `pool_endip`

приклад

```
dhcp_config ={}
dhcp_config.start = "192.168.1.100"
wifi.ap.dhcp.config(dhcp_config)
```

wifi.ap.dhcp.start()

Запускає службу DHCP.

Синтаксис

`wifi.ap.dhcp.start()`

Параметри

немає

Повернення

логічне значення, що вказує на успіх

wifi.ap.dhcp.stop()

Зупиняє службу DHCP.

Синтаксис

`wifi.ap.dhcp.stop()`

Параметри

немає

Повернення

логічне значення, що вказує на успіх

МОДУЛЬ WIFI.EVENTMON

wifi.eventmon.register()

Реєстрація/скасування реєстрації зворотних викликів для монітора подій WiFi. - Після реєстрації зворотного виклику ця функція може бути викликана для оновлення функції зворотного виклику в будь-який час

Примітка

Щоб переконатися, що всі події WiFi перехоплюються, зворотні виклики монітора подій Wi-Fi слід зареєструвати якомога раніше в init.lua. Будь-які події, які відбуваються до реєстрації зворотних викликів, будуть відхилені!

Синтаксис

wifi.eventmon.register(Подія[, функція(T)])

Параметри

Подія: подія Wi-Fi, для якої ви хочете встановити зворотній виклик.

- Дійсні події WiFi:
  - wifi.eventmon.STA_CONNECTED
  - wifi.eventmon.STA_DISCONNECTED
  - wifi.eventmon.STA_AUTHMODE_CHANGE
  - wifi.eventmon.STA_GOT_IP
  - wifi.eventmon.STA_DHCP_TIMEOUT
  - wifi.eventmon.AP_STACONNECTED
  - wifi.eventmon.AP_STADISCONNECTED
  - wifi.eventmon.AP_PROBEREQRECVED

Повернення

функція: `nil`

Зворотний виклик: T: Таблиця, повернута подією.

- `wifi.eventmon.STA_CONNECTED` Станція підключена до точки доступу.
  - `SSID` : SSID точки доступу.

    - `BSSID` : BSSID точки доступу.
    - `channel` : канал, на якому працює точка доступу.
- `wifi.eventmon.STA_DISCONNECTED` : станція була відключена від точки доступу.
    - `SSID` : SSID точки доступу.
    - `BSSID` : BSSID точки доступу.
    - `reason` : Див. wifi.eventmon.reason нижче.
- `wifi.eventmon.STA_AUTHMODE_CHANGE` : Точка доступу змінила режим авторизації.
    - `old_auth_mode` : Старий режим авторизації Wi-Fi.
    - `new_auth_mode` : Новий режим авторизації Wi-Fi.
- `wifi.eventmon.STA_GOT_IP` : станція отримала IP-адресу.
    - `IP` : IP-адреса, призначена станції.
    - `netmask` : Маска підмережі.
    - `gateway` : IP-адреса точки доступу, до якої підключена станція.
- `wifi.eventmon.STA_DHCP_TIMEOUT` : Запит станції DHCP минув.
    - Повернено порожню таблицю.
- `wifi.eventmon.AP_STACONNECTED` : новий клієнт підключився до точки доступу.
    - `MAC` : MAC-адреса клієнта, який підключився.
    - `AID` : SDK не надає подробиць щодо цього значення, що повертається.
- `wifi.eventmon.AP_STADISCONNECTED` : Клієнт відключився від точки доступу.

- MAC : MAC-адреса клієнта, який відключився.
- AID : SDK не надає подробиць щодо цього значення, що повертається.

- wifi.eventmon.AP_PROBEREQRECVED : Запит на зондування отримано.
  - MAC : MAC-адреса клієнта, який перевіряє точку доступу.
  - RSSI : Індикатор рівня отриманого сигналу клієнта.
- wifi.eventmon.WIFI_MODE_CHANGE : режим WiFi змінився.
  - old_auth_mode : старий режим WiFi.
  - new_auth_mode : новий режим WiFi.

приклад

```
wifi.eventmon.register(wifi.eventmon.STA_CONNECTED, function(T)
  print("\n\tSTA - CONNECTED".."\n\tSSID: "..T.SSID.."\n\tBSSID: "..
  T.BSSID.."\n\tChannel: "..T.channel)
end)


wifi.eventmon.register(wifi.eventmon.STA_DISCONNECTED, function(T)
  print("\n\tSTA - DISCONNECTED".."\n\tSSID: "..T.SSID.."\n\tBSSID: "..
  T.BSSID.."\n\treason: "..T.reason)
end)

wifi.eventmon.register(wifi.eventmon.STA_AUTHMODE_CHANGE, function(T)
  print("\n\tSTA - AUTHMODE CHANGE".."\n\told_auth_mode: "..
  T.old_auth_mode.."\n\tnew_auth_mode: "..T.new_auth_mode)
end)

wifi.eventmon.register(wifi.eventmon.STA_GOT_IP, function(T)
  print("\n\tSTA - GOT IP".."\n\tStation IP: "..T.IP.."\n\tSubnet mask:
"..
  T.netmask.."\n\tGateway IP: "..T.gateway)
end)

wifi.eventmon.register(wifi.eventmon.STA_DHCP_TIMEOUT, function()
  print("\n\tSTA - DHCP TIMEOUT")
end)
```

```
wifi.eventmon.register(wifi.eventmon.AP_STACONNECTED, function(T)
  print("\n\tAP - STATION CONNECTED".."\n\tMAC: "..T.MAC.."\n\tAID: 
"..T.AID)
end)

wifi.eventmon.register(wifi.eventmon.AP_STADISCONNECTED, function(T)
  print("\n\tAP - STATION DISCONNECTED".."\n\tMAC: "..T.MAC.."\n\tAID: 
"..T.AID)
end)

wifi.eventmon.register(wifi.eventmon.AP_PROBEREQRECVED, function(T)
  print("\n\tAP - PROBE REQUEST RECEIVED".."\n\tMAC: ".. T.MAC.."\n\tRSSI: 
"..T.RSSI)
end)

wifi.eventmon.register(wifi.eventmon.WIFI_MODE_CHANGED, function(T)
  print("\n\tSTA - WIFI MODE CHANGED".."\n\told_mode: "..
  T.old_mode.."\n\tnew_mode: "..T.new_mode)
end)
```

Дивись також

- wifi.eventmon.unregister()

wifi.eventmon.unregister()

Скасовує реєстрацію зворотних викликів для монітора подій WiFi.

Синтаксис

wifi.eventmon.unregister(Подія)

Параметри

Подія: подія WiFi, для якої ви хочете видалити зворотний виклик.

- Дійсні події WiFi:
  - wifi.eventmon.STA_CONNECTED
  - wifi.eventmon.STA_DISCONNECTED
  - wifi.eventmon.STA_AUTHMODE_CHANGE
  - wifi.eventmon.STA_GOT_IP
  - wifi.eventmon.STA_DHCP_TIMEOUT
  - wifi.eventmon.AP_STACONNECTED
  - wifi.eventmon.AP_STADISCONNECTED
  - wifi.eventmon.AP_PROBEREQRECVED

- wifi.eventmon.WIFI_MODE_CHANGED

Повернення

nil

приклад

```
wifi.eventmon.unregister(wifi.eventmon.STA_CONNECTED)
```

Дивись також

- wifi.eventmon.register()

wifi.eventmon.reason

Таблиця з причинами відключення.

Причина відключення	значення
wifi.eventmon.reason.НЕЗАДАНО	1
wifi.eventmon.reason.AUTH_EXPIRE	2
wifi.eventmon.reason.AUTH_LEAVE	3
wifi.eventmon.reason.ASSOC_EXPIRE	4
wifi.eventmon.reason.ASSOC_TOOMMANY	5
wifi.eventmon.reason.NOT_AUTHED	6
wifi.eventmon.reason.NOT_ASSOCED	7
wifi.eventmon.reason.ASSOC_LEAVE	8
wifi.eventmon.reason.ASSOC_NOT_AUTHED	9
wifi.eventmon.reason.DISASSOC_PWRCAP_BAD	10
wifi.eventmon.reason.DISASSOC_SUPCHAN_BAD	11
wifi.eventmon.reason.IE_INVALID	13
wifi.eventmon.reason.MIC_FAILURE	14
wifi.eventmon.reason.4WAY_HANDSHAKE_TIMEOUT	15

Причина відключення	значення
wifi.eventmon.reason.GROUP_KEY_UPDATE_TIMEOUT	16
wifi.eventmon.reason.IE_IN_4WAY_DIFFERS	17
wifi.eventmon.reason.GROUP_CIPHER_INVALID	18
wifi.eventmon.reason.PAIRWISE_CIPHER_INVALID	19
wifi.eventmon.reason.AKMP_INVALID	20
wifi.eventmon.reason.UNSUPP_RSN_IE_VERSION	21
wifi.eventmon.reason.INVALID_RSN_IE_CAP	22
wifi.eventmon.reason.802_1X_AUTH_FAILED	23
wifi.eventmon.reason.CIPHER_SUITE_REJECTED	24
wifi.eventmon.reason.BEACON_TIMEOUT	200
wifi.eventmon.reason.NO_AP_FOUND	201
wifi.eventmon.reason.AUTH_FAIL	202
wifi.eventmon.reason.ASSOC_FAIL	203
wifi.eventmon.reason.HANDSHAKE_TIMEOUT	204

МОДУЛЬ WIFI.MONITOR

Це додатковий модуль, який включається, лише якщо LUA_USE_MODULES_WIFI_MONITOR визначено у user_modules.h файлі. Цей модуль забезпечує доступ до функцій режиму монітора чіпсета ESP8266. Зокрема, він забезпечує доступ до отриманих кадрів керування WiFi.

wifi.monitor.start()	Реєструє функцію зворотного виклику, яка буде викликатися кожного разу, коли отримано керуючий кадр.
wifi.monitor.stop()	Вимкне режим монітора та повернеться до нормальної роботи.
wifi.monitor.channel()	Це встановлює номер каналу для моніторингу.
packet:radio_byte()	
packet:frame_byte()	
packet:radio_sub()	
пакет:frame_sub()	
пакет:radio_subhex()	
пакет:frame_sub()	
packet:ie_table()	Повертає таблицю інформаційних елементів із кадру керування.
пакет.	Об'єкт пакета має багато атрибутів.
Заголовок радіо	Радіо заголовок згадувався вище як 12-байтова структура.

wifi.monitor.start()

Реєструє функцію зворотного виклику, яка буде викликатися кожного разу, коли отримано керуючий кадр.

Повертаються лише перші 110 байт або близько того кадру - це обмеження SDK. Будь-яка підключена точка доступу/станція буде відключена. Виклик цієї функції повертає канал до 1.

Синтаксис

`wifi.monitor.start([filter parameters,] mgmt_frame_callback)`

Параметри

- параметри фільтра. Це зміщення байтів (на основі 1) у базовій структурі даних, значення для порівняння та необов'язкова маска для використання для відповідності. Структура даних, яка використовується для фільтрації, складається з 12 байт радіозаголовка , а потім фактичного кадру. Таким чином, перший байт кадру має номер 13. Значення фільтра 13, 0x80 просто вилучатимуть кадри маяків.
- `mgmt_frame_callback` це функція, яка викликається з одним аргументом, який є `wifi.packet` об'єктом, який має багато методів і атрибутів.

Приклад

```
wifi.monitor.start(13, 0x80, function(pkt)
    print ('Beacon: ' .. pkt.bssid_hex .. " '" .. pkt[0] .. "' ch "
.. pkt[3]:byte(1))
end)
wifi.monitor.channel(6)
```

wifi.monitor.stop()

Вимкне режим монітора та повернеться до нормальної роботи. Немає параметрів і значення, що повертається.

Синтаксис

```
wifi.monitor.stop()
```

wifi.monitor.channel()

Встановлює номер каналу для моніторингу. Кадри маяків (зокрема) зазвичай надсилаються кожні 102 мілісекунди.

Синтаксис

```
wifi.monitor.channel(channel)
```

Параметри

- `channel` встановлює номер каналу в діапазоні від 1 до 15.

Повернення

нічого.

ОБ'ЄКТ wifi.packet

Цей об’єкт забезпечує доступ до необроблених пакетних даних, а також багато методів для простого вилучення даних із пакета.

packet:radio_byte()

Синтаксис

```
packet:radio_byte(n)
```

Параметри

- `n` номер байта (від 1), який потрібно отримати з

  частини радіозаголовка пакета

Повернення

0-255 як значення байта нічого, якщо зміщення не входить

до радіозаголовка .

packet:frame_byte()

Це схоже на `string.byte` метод, за винятком того, що він надає доступ до байтів отриманого кадру.

Синтаксис

```
packet:frame_byte(n)
```

Параметри

- `n` номер байта (від 1), який потрібно отримати з отриманого кадру.

Повернення

0-255 як значення байта нічого, якщо зміщення не в межах отриманого кадру.

packet:radio_sub()

Це схоже на `string.sub` метод, за винятком того, що він надає доступ до байтів радіозаголовка .

Синтаксис

```
packet:radio_sub(start, end)
```

Параметри

Ті самі правила, що й для, `string.sub` за винятком того, що він діє на радіозаголовку .

Повернення

Рядок за `string.sub` правилами.

пакет:frame_sub()

Це схоже на string.sub метод, за винятком того, що він надає доступ до байтів отриманого кадру.

Синтаксис

packet:frame_sub(start, end)

Параметри

Ті самі правила, що й для, string.sub за винятком того, що він діє на отриманий кадр.

Повернення

Рядок за string.sub правилами.

пакет:radio_subhex()

Це схоже на string.sub метод, за винятком того, що він надає доступ до байтів радіозаголовка . Він також ефективно перетворює їх на hex.

Синтаксис

packet:radio_subhex(start, end [, seperator])

Параметри

Ті самі правила, що й для, string.sub за винятком того, що він діє на радіозаголовку . - seperator це необов'язкове жало, яке розміщується між окремими повернутими парами гексів.

Повернення

Рядок за string.sub правилами, перетворена в шестигранну з можливими вставленими прокладками.

packet:ie_table()

Це повертає таблицю інформаційних елементів із кадру керування. Значеннями ключів таблиці є номери елементів інформації (0 - 255). Зверніть увагу, що IE0 є SSID. Цей метод здебільшого корисний, лише якщо вам потрібно визначити, які інформаційні елементи були у кадрі керування.

Синтаксис

```
packet:ie_table()
```

Параметри

Жодного.

Повернення

Таблиця з усіма інформаційними елементами в ній.

приклад

```
print ("SSID", packet:ie_table()[0])
```

Альтернатива

Сам об’єкт packet може бути проіндексований для вилучення інформаційних елементів.

приклад

```
print ("SSID", packet[0])
```

пакет.<атрибут>

Об'єкт пакета має багато атрибутів. Вони забезпечують легкий доступ до всіх полів, хоча й непростий спосіб їх перерахування. Усі цілі числа є беззнаковими, крім зазначених.

Інформаційні елементи повертаються, лише якщо вони повністю знаходяться в межах захопленого кадру. Це може означати, що для деяких кадрів можуть бути відсутні деякі інформаційні елементи.

Коли рядок повертається як значення поля, він може бути двійковим рядком із вбудованими нулями.

Усі інформаційні елементи повертаються як рядки, навіть якщо вони мають довжину лише один байт і виглядають як число в специфікації.

Це виключно для узгодженості інтерфейсу. Зауважте, що навіть SSID можуть містити вбудовані нулі.

Attribute name	**Type**
aggregation	Integer
ampdu_cnt	Integer
association_id	Integer
authentication_algorithm	Integer
authentication_transaction	Integer
beacon_interval	Integer
beacon_interval	Integer
bssid	String
bssid_hex	String
bssidmatch0	Integer
bssidmatch1	Integer
capability	Integer
channel	Integer
current_ap	String
cwb	Integer
dmatch0	Integer
dmatch1	Integer
dstmac	String
dstmac_hex	String
duration	Integer

Attribute name	Type
fec_coding	Integer
frame	String
frame_hex	String
fromds	Integer
header	String
ht_length	Integer
ie_20_40_bss_coexistence	String
ie_20_40_bss_intolerant_channel_report	String
ie_advertisement_protocol	String
ie_aid	String
ie_antenna	String
ie_ap_channel_report	String
ie_authenticated_mesh_peering_exchange	String
ie_beacon_timing	String
ie_bss_ac_access_delay	String
ie_bss_available_admission_capacity	String
ie_bss_average_access_delay	String
ie_bss_load	String
ie_bss_max_idle_period	String
ie_cf_parameter_set	String
ie_challenge_text	String
ie_channel_switch_announcement	String
ie_channel_switch_timing	String

Attribute name	Type
ie_channel_switch_wrapper	String
ie_channel_usage	String
ie_collocated_interference_report	String
ie_congestion_notification	String
ie_country	String
ie_destination_uri	String
ie_diagnostic_report	String
ie_diagnostic_request	String
ie_dms_request	String
ie_dms_response	String
ie_dse_registered_location	String
ie_dsss_parameter_set	String
ie_edca_parameter_set	String
ie_emergency_alart_identifier	String
ie_erp_information	String
ie_event_report	String
ie_event_request	String
ie_expedited_bandwidth_request	String
ie_extended_bss_load	String
ie_extended_capabilities	String
ie_extended_channel_switch_announcement	String
ie_extended_supported_rates	String
ie_fast_bss_transition	String

Attribute name	Type
ie_fh_parameter_set	String
ie_fms_descriptor	String
ie_fms_request	String
ie_fms_response	String
ie_gann	String
ie_he_capabilities	String
ie_hopping_pattern_parameters	String
ie_hopping_pattern_table	String
ie_ht_capabilities	String
ie_ht_operation	String
ie_ibss_dfs	String
ie_ibss_parameter_set	String
ie_interworking	String
ie_link_identifier	String
ie_location_parameters	String
ie_management_mic	String
ie_mccaop	String
ie_mccaop_advertisement	String
ie_mccaop_advertisement_overview	String
ie_mccaop_setup_reply	String
ie_mccaop_setup_request	String
ie_measurement_pilot_transmission	String
ie_measurement_report	String

Attribute name	Type
ie_measurement_request	String
ie_mesh_awake_window	String
ie_mesh_channel_switch_parameters	String
ie_mesh_configuration	String
ie_mesh_id	String
ie_mesh_link_metric_report	String
ie_mesh_peering_management	String
ie_mic	String
ie_mobility_domain	String
ie_multiple_bssid	String
ie_multiple_bssid_index	String
ie_neighbor_report	String
ie_nontransmitted_bssid_capability	String
ie_operating_mode_notification	String
ie_overlapping_bss_scan_parameters	String
ie_perr	String
ie_power_capability	String
ie_power_constraint	String
ie_prep	String
ie_preq	String
ie_proxy_update	String
ie_proxy_update_confirmation	String
ie_pti_control	String

Attribute name	Type
ie_qos_capability	String
ie_qos_map_set	String
ie_qos_traffic_capability	String
ie_quiet	String
ie_quiet_channel	String
ie_rann	String
ie_rcpi	String
ie_request	String
ie_ric_data	String
ie_ric_descriptor	String
ie_rm_enabled_capacities	String
ie_roaming_consortium	String
ie_rsn	String
ie_rsni	String
ie_schedule	String
ie_secondary_channel_offset	String
ie_ssid	String
ie_ssid_list	String
ie_supported_channels	String
ie_supported_operating_classes	String
ie_supported_rates	String
ie_tclas	String
ie_tclas_processing	String

Attribute name	Type
ie_tfs_request	String
ie_tfs_response	String
ie_tim	String
ie_tim_broadcast_request	String
ie_tim_broadcast_response	String
ie_time_advertisement	String
ie_time_zone	String
ie_timeout_interval	String
ie_tpc_report	String
ie_tpc_request	String
ie_tpu_buffer_status	String
ie_ts_delay	String
ie_tspec	String
ie_uapsd_coexistence	String
ie_vendor_specific	String
ie_vht_capabilities	String
ie_vht_operation	String
ie_vht_transmit_power_envelope	String
ie_wakeup_schedule	String
ie_wide_bandwidth_channel_switch	String
ie_wnm_sleep_mode	String
is_group	Integer
legacy_length	Integer

Attribute name	Type
listen_interval	Integer
mcs	Integer
moredata	Integer
moreflag	Integer
not_counding	Integer
number	Integer
order	Integer
protectedframe	Integer
protocol	Integer
pwrmgmt	Integer
radio	String
rate	Integer
reason	Integer
retry	Integer
rssi	Signed Integer
rxend_state	Integer
sgi	Integer
sig_mode	Integer
smoothing	Integer
srcmac	String
srcmac_hex	String
status	Integer
stbc	Integer

Attribute name	Type
subtype	Integer
timestamp	String
tods	Integer
type	Integer

приклад

```
print ("SSID", packet.ie_ssid)
```

МОДУЛЬ АЦП

Модуль АЦП забезпечує доступ до вбудованого АЦП.

На ESP8266 є тільки одноканальний, який мультиплексується з напругою акумулятора. Залежно від налаштування в «esp init data» (байт 107) можна або використовувати АЦП для зчитування зовнішньої напруги, або для зчитування напруги системи (vdd33), але не обидва.

У якому режимі використовувати АЦП, можна налаштувати за допомогою `adc.force_init_mode()` функції. Зауважте, що після переходу з одного на інший потрібен перезапуск системи (наприклад, цикл живлення, кнопка скидання, `node.restart()`), перш ніж зміни набудуть чинності.

adc.force_init_mode()	Перевіряє та, якщо необхідно, переконфігурує налаштування режиму АЦП у блоці початкових даних ESP.
adc.read()	Зчитує АЦП.
adc.readvdd33()	Зчитує напругу системи.

ОПИС ФУНКЦІЙ API МОДУЛЯ АЦП

adc.force_init_mode()

Перевіряє та, якщо необхідно, переконфігурує налаштування режиму АЦП у блоці початкових даних ESP.

Синтаксис

`adc.force_init_mode(mode_value)`

Параметри

`mode_value` Один із `adc.INIT_ADC` або `adc.INIT_VDD33`.

Повернення

True, якщо функція мала змінити режим, false, якщо режим уже налаштовано. У разі справжнього повернення ESP потрібно перезапустити, щоб зміни набули чинності.

приклад

```
-- in you init.lua:
if adc.force_init_mode(adc.INIT_VDD33)
then
  node.restart()
  return -- don't bother continuing, the restart is scheduled
end

print("System voltage (mV):", adc.readvdd33(0))
```

Дивись також

`node.restart()`

adc.read()

Зразки АЦП.

Синтаксис

`adc.read(channel)`

Параметри

`channel` завжди 0 на ESP8266

Повернення

вибіркове значення (число)

Якщо ESP8266 налаштовано на використання АЦП для зчитування напруги системи, ця функція завжди повертатиме 65535. Це обмеження апаратного забезпечення та/або SDK.

Приклад

```
val = adc.read(0)
```

adc.readvdd33()

Зчитує напругу ADC.

Синтаксис

```
adc.readvdd33()
```

Параметри

немає

Повернення

напруга в мілівольтах (число)

Якщо ESP8266 налаштовано на використання АЦП для вибірки зовнішнього контакту, ця функція завжди повертатиме 65535. Це обмеження апаратного забезпечення та/або SDK.

МОДУЛЬ PWM

pwm.close()	Quit PWM mode for the specified GPIO pin.
pwm.getclock()	Get selected PWM frequency of pin.
pwm.getduty()	Get selected duty cycle of pin.
pwm.setclock()	Set PWM frequency.
pwm.setduty()	Set duty cycle for a pin.
pwm.setup()	Set pin to PWM mode.
pwm.start()	PWM starts, the waveform is applied to the GPIO pin.
pwm.stop()	Pause the output of the PWM waveform.

ОПИС ФУНКЦІЙ API МОДУЛЯ PWM

pwm.close()

Вийти з режиму PWM для вказаного контакту GPIO.

Синтаксис

```
pwm.close(pin)
```

Параметри

`pin` 1~12, індекс IO

Повернення

`nil`

Дивись також

pwm.start()

pwm.getclock()

Отримати вибрану частоту ШІМ контакту.

Синтаксис

`pwm.getclock(pin)`

Параметри

`pin` 1~12, індекс IO

Повернення

`number` ШІМ частота виводу

Дивись також

pwm.setclock()

Дивись також

pwm.getduty()

pwm.getduty()

Отримати вибраний робочий цикл піна.

Синтаксис

`pwm.getduty(pin)`

Параметри

`pin` 1~12, індекс IO

Повернення

`number` робочий цикл, макс. 1023

Дивись також

pwm.setduty()

pwm.setclock()

Встановіть частоту ШІМ.

Примітка. Налаштування частоти ШІМ синхронно змінить інші налаштування, якщо такі є. Для системи можна дозволити лише одну частоту ШІМ.

Синтаксис

`pwm.setclock(pin, clock)`

Параметри

- `pin` 1~12, індекс IO
- `clock` 1~1000, частота ШІМ

Повернення

`nil`

Дивись також

pwm.getclock()

pwm.setduty()

Встановити робочий цикл для піна.

Синтаксис

`pwm.setduty(pin, duty)`

Параметри

- `pin` 1~12, індекс IO
- `duty` 0~1023, робочий цикл ШІМ, макс. 1023 (10 біт)

Повернення

`nil`

приклад

```
-- D1 is connected to green led
-- D2 is connected to blue led
-- D3 is connected to red led
pwm.setup(1, 500, 512)
pwm.setup(2, 500, 512)
pwm.setup(3, 500, 512)
pwm.start(1)
pwm.start(2)
pwm.start(3)
function led(r, g, b)
    pwm.setduty(1, g)
    pwm.setduty(2, b)
```

```
    pwm.setduty(3, r)
end
led(512, 0, 0) --  set led to red
led(0, 0, 512) -- set led to blue.
```

pwm.setup()

Встановіть пін у режим ШІМ. Тільки 6 контактів можна встановити в режим ШІМ.

Синтаксис

`pwm.setup(pin, clock, duty)`

Параметри

- `pin` 1~12, індекс IO
- `clock` 1~1000, частота ШІМ
- `duty` 0~1023, робочий цикл ШІМ, макс. 1023 (10 біт)

Повернення

`nil`

приклад

```
-- set pin index 1 as pwm output, frequency is 100Hz, duty cycle is half.
pwm.setup(1, 100, 512)
```

Дивись також

pwm.start()

pwm.start()

Запускається ШІМ, сигнал подається на контакт GPIO.

Синтаксис

`pwm.start(pin)`

Параметри

`pin` 1~12, індекс IO

Повернення

`nil`

Дивись також

pwm.stop()

pwm.stop()

Призупинити вихід сигналу ШІМ.

Синтаксис

`pwm.stop(pin)`

Параметри

`pin` 1~12, індекс IO

Повернення

`nil`

Дивись також

pwm.start()

МОДУЛЬ PWM2

Модуль для генерації ШІМ-імпульсів на будь-який з контактів GPIO.

ШІМ генерується програмним забезпеченням за допомогою програмного переривання TIMER1 FRC1. Цей модуль використовує таймер в ексклюзивному режимі.

Підтримувані частоти приблизно від 120 кГц (з навантаженням 50%) до імпульсу/53 с (або 250 кГц і 26 с для CPU160). Підтримуються також частки частоти навіть для цілочисельних збірок мікропрограм.

Підтримуються всі контакти GPIO, крім контакту 0.

Можна генерувати різні сигнали ШІМ для будь-якого з них одночасно.

Цей модуль підтримує CPU 80 МГц, а також CPU 160 МГц. Частотні межі однакові, але використовуючи CPU 160 МГц, можна сподіватися, що тим часом ви зможете виконувати більше роботи.

Типове використання таке:

```
pwm2.setup_pin_hz(3,250000,2,1) -- pin 3, PWM freq of 250kHz, pulse period of 2 steps, initial duty 1 period step
pwm2.setup_pin_hz(4,1,2,1) -- pin 4, PWM freq of 1Hz, pulse period of 2 steps, initial duty 1 period step
pwm2.start() -- starts pwm, internal led will blink with 0.5sec interval
...
pwm2.set_duty(4, 2) -- led full off (pin is high)
...
pwm2.set_duty(4, 0) -- led full on (pin is low)
...
pwm2.stop() -- PWM stopped, gpio pin released, timer1 released
```

Understand frequencies	Всі частоти і періоди всередині таймера виражаються в тіках процесора за формулою: cpuTicksPerSecond / (частота Гц * період).
Frequency precision and limits	TIMER1 FRC1 ESP працює на фіксованій власній частоті 5 МГц.

Working with multiple frequencies	При роботі з декількома виводами цей модуль автоматично визначає правильну базову частоту переривання.
Understanding timer use	Цей модуль використовує м'яке переривання TIMER1 FRC1 для генерації ШІМ-сигналу.
Troubleshooting watchdog timeouts	Сторожове переривання зазвичай відбувається, якщо вибрана частота (і період) занадто велика і.
Differences with PWM module	PWM і PWM2 - це модулі, що виконують схожу роботу і мають багато спільного.
pwm2.setup_pin_hz()	Призначає частоту ШІМ, виражену у Гц, для даного виводу.
pwm2.setup_pin_sec()	Призначає частоту ШІМ, виражену як один імпульс за секунду (секунди), для даного виводу.
pwm2.start()	Запускає ШІМ для всіх виводів.
pwm2.stop()	Зупиняє ШІМ для всіх виводів.
pwm2.set_duty()	Встановлює робочий цикл для одного або декількох виводів.
pwm2.release_pin()	Звільняє даний вивід від попередньо виконаних налаштувань.
pwm2.get_timer_data()	Друкує внутрішні структури даних, пов'язані з таймером.
pwm2.get_pin_data()	Друкує внутрішні структури даних, пов'язані із заданим виводом GPIO.

Частоти

Усі внутрішні частоти та періоди виражаються як такти ЦП за такою формулою: `cpuTicksPerSecond / (frequencyHz * period)`. Наприклад, 1 кГц з періодом 1000 для ЦП 80 МГц призводить до 80 тактів ЦП за період, тобто період становить 1 мкс.

Щоб забезпечити кращу настройку, я додав необов'язковий аргумент FrequencyDivisor під час встановлення кеглів. З його допомогою можна виразити частоту як ділення між двома значеннями: `frequency / divisor`. Наприклад, для моделювання частоти 100,1 Гц потрібно вказати частоту 1001 і дільник 10.

Простий спосіб виразити частоти нижче Гц, тобто частоти, які потребують секунд для завершення одного імпульсу, полягає в застосуванні методів налаштування в секундах. Для них формула для обчислення тактів ЦП така `cpuTicksPerSecond * frequencySec / period`.

Максимальна тривалість імпульсу обмежена зміною 32-бітного лічильника тактів внутрішнього ЦП ESP. Для CPU80 це буде кожні 53 секунди, для CPU160 це буде половина.

Точність і межі частоти

TIMER1 FRC1 ESP працює на фіксованій власній частоті 5 МГц. Тому точність окремого переривання становить 200 нс. Але ця межа не може бути досягнута.

Сам код обробника переривань таймера ОС має внутрішні накладні витрати. Для автоматично завантажених переривань це приблизно 50 CPUTick. На короткий проміжок часу можна переривати приблизно на 1 МГц, але потім втрутиться сторожовий таймер.

Власний обробник переривань PWM2 має накладні витрати 162 CPUTick + 12 CPUTick на кожен використаний контакт.

З найшвидшим налаштуванням, тобто 1 контактом, 50% робочим циклом (період імпульсу 2) і CPU80, можна очікувати досягнення частоти ШІМ 125 кГц. Для 12 контактів це впаде приблизно до 100 кГц. З CPU160 можна досягти 220 кГц за допомогою 1 контакту.

Внутрішні частоти виражаються спочатку як такти ЦП, а потім як такти ТАЙМЕРА1. Оскільки частота ТАЙМЕРА1 становить 1/16 частоти ЦП, деяка

точність частоти втрачається під час перетворення з тактів ЦП на ТАЙМЕР. Можна перевірити точні значення, які використовуються через pwm2.get_timer_data() .

Значення interruptTimerCPUTicks представляє бажаний період переривання в CPUTicks.

Значення interruptTimerTicks представляє фактично використаний період переривання як такти TIMER1 (1/16 ЦП).

Робота з кількома частотами

Під час роботи з декількома контактами цей модуль автоматично виявляє правильну базову частоту переривання. Це робиться шляхом обчислення найбільшого спільного дільника частоти та використання його як загальної частоти для всіх контактів.

При використанні однакової частоти для багатьох виводів частоти налаштування одного виводу достатньо для забезпечення точності.

Використовуючи різні частоти, слід звернути особливу увагу на їхній найбільший спільний дільник, виражений як такти ЦП.

Наприклад, змішування 100 кГц з періодом 2 і 0,5 Гц з періодом 2 призводить до основного періоду переривання 800 тактів ЦП. Але зміна на 100 кГц+1 легко призведе до дільника 1. Це явно неробоча комбінація.

Іншим прикладом є частота 120 кГц з періодом 2, що призводить до періоду 333 тактів ЦП. У поєднанні з парною результуючою частотою, як-от 1 Гц з періодом 2, це призведе до загального дільника 1, що, очевидно, також є неробочим налаштуванням.

Використання таймера

Цей модуль використовує програмне переривання TIMER1 FRC1 для генерації сигналу ШІМ. Оскільки його переривання можуть бути замасковані, оскільки це робить якась частина ОС, можна мати певний вплив на якість генерованого ШІМ-сигналу.

Як загальний принцип, не слід очікувати високої точності сигналу від цього модуля. Також зауважте, що маскування переривання залежить від інших дій, що відбуваються в ESP, крім модуля pwm2.

Крім того, цей таймер використовується іншими модулями, такими як pwm, pcm, ws2812 тощо. Оскільки на таймері підтримується ексклюзивне блокування, одночасне використання таких модулів буде неможливим.

Усунення несправностей сторожового тайм-ауту

Сторожове переривання зазвичай виникає, якщо вибрана частота (і період) занадто велика, тобто занадто мале значення тактів таймера. Для CPU80 МГц, я вважаю, поріг становить приблизно 125 кГц з періодом 2 і одним контактом (CPU80), враховуючи невелике інше навантаження на систему. Для CPU160 поріг становить 225 кГц.

Іншою причиною виникнення сторожового переривання є змішування не дуже сумісних частот, коли використовується кілька контактів.

Відмінності з модулем ШІМ

- PWM і PWM2 - це модулі, які виконують однакову роботу і мають багато спільного. Ось кілька основних моментів PWM2 у порівнянні з модулем PWM:
- PWM2 використовує винятково TIMER1, що забезпечує, можливо, кращу якість сигналу PWM

- PWM2 може генерувати частоти ШІМ в діапазоні від 1 імпульсу/53 секунди до 125 кГц (26 секунд/225 кГц для CPU160)
- ШІМ2 може генерувати частоти ШІМ із частками, тобто 1001 кГц
- PWM2 підтримує CPU160
- PWM2 підтримує практично всі порти GPIO одночасно

На відміну від PWM2, PWM може:

- генерувати ШІМ-імпульс із трохи більшим робочим циклом, тобто 1 кГц за періоду 1000 імпульсів
- можна використовувати одночасно з деякими іншими модулями, наприклад gpio.pulse

ОПИС ФУНКЦІЙ API МОДУЛЯ PWM

pwm2.setup_pin_hz()

Призначає частоту ШІМ, виражену в Гц, на даний контакт. Цей метод підходить для встановлення частот у діапазоні >= 1 Гц.

Синтаксис

`pwm2.setup_pin_hz(pin,frequencyAsHz,pulsePeriod,initialDuty`

`[,frequencyDivisor])`

Параметри

- `pin` 1-12
- `frequencyAsHz` бажана частота в Гц, наприклад 1000 для 1 кГц
- `pulsePeriod` дискретні кроки в одному імпульсі ШІМ, наприклад 100
- `initialDuty` початкове навантаження з кроками періоду імпульсу, тобто 50 для 50% імпульсу з роздільною здатністю 100

- `frequencyDivisor` ціле число для ділення добутку частоти та pulsePeriod. Використовується для формування частотних часток. За замовчуванням не потрібно.

Повернення

`nil`

Дивись також

- pwm2.setup_pin_sec()
- pwm2.start()
- pwm2.release_pin()
- розуміння частот
- робота з кількома частотами
- pwm2.get_timer_data()

pwm2.setup_pin_sec()

Призначає частоту ШІМ, виражену як один імпульс на секунду(и), на даний контакт. Цей метод підходить для встановлення частот у діапазоні 0 < 1 Гц, але замість цього виражається у секундах. Наприклад, 0,5 Гц виражаються як 2-секундний імпульс.

Синтаксис

`pwm2.setup_pin_sec(pin,frequencyAsSec,pulsePeriod,initialDuty`

`[,frequencyDivisor])`

Параметри

- `pin` 1-12
- `frequencyAsSec` бажана частота як один імпульс протягом заданих секунд, наприклад 2 означає ШІМ з імпульсом довжиною 2 секунди.
- `pulsePeriod` дискретні кроки в одному імпульсі ШІМ, наприклад 100

- `initialDuty` початкове навантаження з кроками періоду імпульсу, тобто 50 для 50% імпульсу з роздільною здатністю 100
- `frequencyDivisor` ціле число для ділення добутку частоти та pulsePeriod. Використовується для формування частотних часток. За замовчуванням не потрібно.

Повернення

`nil`

Дивись також

- pwm2.setup_pin_hz()
- pwm2.start()
- pwm2.release_pin()
- розуміння частот
- робота з кількома частотами
- pwm2.get_timer_data()

pwm2.start()

Запускає ШІМ для всіх контактів налаштування. На даний момент контакти GPIO позначені як вихідні, а TIMER1 зарезервовано для цього модуля. Якщо TIMER1 вже зарезервований іншим модулем, цей метод повідомляє про помилку Lua та повертає false.

Синтаксис

`pwm2.start()`

Параметри

`nil`

Повернення

- `bool` true, якщо ШІМ почався нормально, false для TIMER1 зарезервовано іншим модулем.

Дивись також

- pwm2.setup_pin_hz()
- pwm2.setup_pin_sec()
- pwm2.set_duty()
- pwm2.stop()

pwm2.stop()

Зупиняє ШІМ для всіх контактів. Усі контакти GPIO та TIMER1 звільняються. Можна відновити ШІМ з попередніми налаштуваннями контактів, викликавши pwm2.start() одразу після зупинки.

Синтаксис

```
pwm2.stop()
```

Параметри

nil

Повернення

nil

Дивись також

- pwm2.start()
- pwm2.release_pin()

pwm2.set_duty()

Встановлює робочий цикл для одного або кількох контактів. Цей метод миттєво впливає на поточну генерацію ШІМ.

Синтаксис

```
pwm2.set_duty(pin, duty [,pin,duty]*)
```

Параметри

- pin 1~12, індекс IO
- duty 0~період, робочий цикл ШІМ

Повернення

`nil`

Дивись також

- pwm2.stop()

pwm2.release_pin()

Вивільняє вказаний PIN-код із попереднього налаштування. Цей метод застосовний, коли ШІМ зупинено і даний пін більше не потрібен. Вивільняючі піни не обов'язкові. Цей метод корисний для ситуацій старт-стоп-старт, коли кегли змінюються.

Синтаксис

`pwm2.release_pin(pin)`

Параметри

- `pin` 1~12, індекс IO

Повернення

`nil`

Дивись також

- pwm2.setup_pin_hz()
- pwm2.setup_pin_sec()
- pwm2.stop()

pwm2.get_timer_data()

Друкує внутрішні структури даних, пов’язані з таймером. Цей метод корисний для людей, які вирішують проблеми з частотою побічних ефектів.

Синтаксис

`pwm2.get_timer_data()`

Параметри

`nil`

Повернення

- `isStarted` bool, якщо був запущений справжній PWM2
- `interruptTimerCPUTicks` int, бажаний період переривання таймера в тактах ЦП
- `interruptTimerTicks` int, фактичний період переривання таймера в тиках таймера

приклад

```
isStarted, interruptTimerCPUTicks, interruptTimerTicks =
pwm2.get_timer_data()
```

Дивись також

- pwm2.setup_pin_hz()
- pwm2.setup_pin_sec()
- pwm2.get_pin_data()

pwm2.get_pin_data()

Друкує внутрішні структури даних, пов'язані з заданим контактом GPIO. Цей метод корисний для людей, які вирішують проблеми з частотою побічних ефектів.

Синтаксис

`pwm2.get_pin_data(pin)`

Параметри

- `pin` 1~12, індекс IO

Повернення

- `isPinSetup` bool, якщо встановлено 1 контакт
- `duty` внутр., призначений обов
- `pulseResolutions` int, призначені періоди пульсу
- `divisableFrequency` int, призначена частота

- frequencyDivisor int, призначений дільник частоти
- resolutionCPUTicks int, розрахований один період імпульсу в тактах ЦП
- resolutionInterruptCounterMultiplier int, скільки переривань таймера складають один період імпульсу

приклад

```
isPinSetup, duty, pulseResolutions, divisableFrequency, frequencyDivisor,
resolutionCPUTicks, resolutionInterruptCounterMultiplier =
pwm2..get_pin_data(4)
```

Дивись також

- pwm2.setup_pin_hz()
- pwm2.setup_pin_sec()
- pwm2.get_timer_data()

МОДУЛЬ UART

Модуль UART (універсальний асинхронний приймач/передавач) дозволяє конфігурувати та спілкуватися через послідовний порт UART.

Налаштування за замовчуванням для uart контролюється налаштуваннями часу збірки. Швидкість за замовчуванням становить 115 200 біт/с.

Крім того, автоматичне визначення швидкості передачі даних увімкнено протягом перших двох хвилин після завантаження платформи.

Це призведе до перемикання на правильну швидкість передачі після отримання кількох символів. Автоматичне визначення швидкості передачі даних вимкнено під час `uart.setup` виклику.

uart.alt()	Змінити призначення контактів UART.
uart.on()	Встановлює функцію зворотного виклику для обробки подій UART.
uart.setup()	(Повторно) налаштовує параметри зв’язку UART.
uart.getconfig()	Повертає поточні параметри конфігурації UART.
uart.write()	Записати рядок або байт в UART.
uart.fifodepth()	Повідомте про глибину в байтах апаратних черг TX або RX, пов’язаних з UART.

ОПИС ФУНКЦІЙ API МОДУЛЯ UART

uart.alt()

Змінити призначення контактів UART.

Синтаксис

`uart.alt(on)`

Параметри

`on`

- 0 для стандартних пінів

- 1 для використання альтернативних контактів GPIO13 і GPIO15

Повернення

`nil`

uart.on()

Встановлює функцію зворотного виклику для обробки подій UART.

Синтаксис

`uart.on(method, [number/end_char], [function], [run_input])`

Параметри

- `method` "дані", дані отримано на UART
- `number/end_char`
  - якщо n=0, отримає кожен символ у буфері
  - якщо n<255, зворотний виклик викликається, коли отримано n символів
  - якщо один символ "c", зворотній виклик буде викликано, коли зустрінеться "c", або максимум n=255 отримано
- `function` функція зворотного виклику, подія "data" має такий зворотний виклик: `function(data) end`
- `run_input` 0 або 1. Якщо 0, вхідні дані з UART не надходитимуть до інтерпретатора Lua, і він може приймати двійкові дані. Якщо 1, вхідні дані від UART розглядаються як текстовий потік із символами `DEL`, і `BS`, які обробляються як зазвичай. Завершені рядки будуть передані інтерпретатору Lua для виконання. *Зауважте, що інтерпретатор обробляє лише повні рядки.* `CRLF`

Щоб скасувати реєстрацію зворотного виклику, вкажіть лише параметр «data».

Повернення

`nil`

приклад

```
-- when 4 chars is received.
uart.on("data", 4,
  function(data)
    print("receive from uart:", data)
    if data=="quit" then
      uart.on("data") -- unregister callback function
    end
end, 0)
-- when '\r' is received.
uart.on("data", "\r",
  function(data)
    print("receive from uart:", data)
    if data=="quit\r" then
      uart.on("data") -- unregister callback function
    end
end, 0)
```

uart.setup()

Налаштовує параметри зв'язку UART.

Примітка

Байти, надіслані до UART, можуть бути втрачені, якщо ця функція переконфігурує UART під час прийому.

Синтаксис

`uart.setup(id, baud, databits, parity, stopbits[, echo])`

Параметри

- `id` Ідентифікатор UART (0 або 1).
- `baud` один із 300, 600, 1200, 2400, 4800, 9600, 19200, 31250, 38400, 57600, 74880, 115200, 230400, 256000, 460800, 921600, 1843200, 3686400
- `databits` один із 5, 6, 7, 8

- `parity` `uart.PARITY_NONE`, `uart.PARITY_ODD`, або `uart.PARITY_EVEN`
- `stopbits` `uart.STOPBITS_1`, `uart.STOPBITS_1_5`, або `uart.STOPBITS_2`
- `echo` якщо 0, вимкнути відлуння, інакше увімкнути відлуння (за замовчуванням, якщо опущено)

Повернення

налаштована швидкість передачі даних (кількість)

приклад

```
-- configure for 9600, 8N1, with echo
uart.setup(0, 9600, 8, uart.PARITY_NONE, uart.STOPBITS_1, 1)
```

uart.getconfig()

Повертає поточні параметри конфігурації UART.

Синтаксис

`uart.getconfig(id)`

Параметри

- `id` Ідентифікатор UART (0 або 1).

Повернення

Чотири значення:

- `baud` один із 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 74880, 115200, 230400, 256000, 460800, 921600, 1843200, 3686400
- `databits` один із 5, 6, 7, 8
- `parity` `uart.PARITY_NONE`, `uart.PARITY_ODD`, або `uart.PARITY_EVEN`
- `stopbits` `uart.STOPBITS_1`, `uart.STOPBITS_1_5`, або `uart.STOPBITS_2`

приклад

```
print (uart.getconfig(0))
-- prints 9600 8 0 1   for 9600, 8N1
```

uart.write()

Записати рядок або байт в UART.

Синтаксис

`uart.write(id, data1 [, data2, ...])`

Параметри

- `id` Ідентифікатор UART (0 або 1).
- `data1` ... рядок або байт для надсилання через UART

Повернення

`nil`

приклад

```
uart.write(0, "Hello, world\n")
```

uart.fifodepth()

Повідомлення про глибину в байтах апаратних черг TX або RX, пов'язаних з UART.

Синтаксис

`uart.fifodepth(id, dir)`

Параметри

- `id` Ідентифікатор UART (0 або 1).
- `dir` `uart.DIR_RX` для RX FIFO, `uart.DIR_TX` для TX FIFO.

Повернення

Кількість байтів у вибраному FIFO.

МОДУЛЬ SOFTUART

Модуль SoftUART пропонує бітові послідовні порти через контакти GPIO.

ESP8266 має лише 1 повний апаратний порт UART, який використовується для програмування чіпа та зв'язку з мікропрограмою NodeMCU.

Другий порт призначений лише для передачі. Цей модуль забезпечує доступ до більшої кількості портів UART і може використовуватися для зв'язку з такими пристроями, як модулі GSM або GPS.

Наразі не підтримує інвертовану послідовну логіку даних або інші режими, окрім 8N1. Важливо зауважити, що це програмна реалізація послідовного протоколу. Можуть виникнути деякі переривання, які спричиняють збій передавання або отримання через неправильний час.

Примітка

SoftUART не можна використовувати на контакті D0.

softuart.setup()	Створює новий екземпляр SoftUART.
softuart.port:on()	Налаштовує функцію зворотного виклику для отримання даних.
softuart.port:write()	Передає байт або їх послідовність.

ОПИС ФУНКЦІЙ API МОДУЛЯ SOFTUART

softuart.setup()

Створює новий екземпляр SoftUART. Зауважте, що пін rx не може бути спільним для екземплярів, але контакт tx може.

Синтаксис

`softuart.setup(baudrate, txPin, rxPin)`

Параметри

- `baudrate`: швидкість передачі даних SoftUART. Максимально підтримується 230400.
- `txPin`: контакт SoftUART tx. Якщо встановлено `nil` `write` метод, не підтримуватиметься.
- `rxPin`: контакт SoftUART rx. Якщо встановлено `nil` `on("data")` метод, не підтримуватиметься.

Повернення

`softuart` екземпляр.

приклад

```
-- Create new software UART with baudrate of 9600, D2 as Tx pin and D3 as
Rx pin
s = softuart.setup(9600, 2, 3)
```

ПОРТ SOFTUART

softuart.port:on()

Налаштовує функцію зворотного виклику для отримання даних.

Синтаксис

`softuart.port:on(event, trigger, function(data))`

Параметри

- `event` : назва події. Наразі `data` підтримується лише.
- `trigger` : може бути символом або числом. Якщо встановлено символ, функція зворотного виклику запускатиметься лише тоді, коли цей символ буде отримано. Коли встановлено число, функція зворотного виклику запускатиметься лише тоді, коли буфер матиме стільки символів, скільки число.
- `function(data)` : функція зворотного виклику. параметром `data` є програмний буфер прийому UART.

Повернення

`nil`

приклад

```
-- Create new software UART with baudrate of 9600, D2 as Tx pin and D3 as
Rx pin
s = softuart.setup(9600, 2, 3)
-- Set callback to run when 10 characters show up in the buffer
s:on("data", 10, function(data)
  print("Lua handler called!")
  print(data)
end)
```

softuart.port:write()

Передає байт або їх послідовність.

Синтаксис

`softuart.port:write(data)`

Параметри

- `data` : може бути числом або рядком. Коли передається число, буде надіслано лише один байт. Коли передається рядок, буде передана вся послідовність.

Повернення

`nil`

приклад

```
-- Create new software UART with baudrate of 9600, D2 as Tx pin and D3 as
Rx pin
s = softuart.setup(9600, 2, 3)
s:write("Hello!")
-- Send character 'a'
s:write(97)
```

МОДУЛЬ ТАЙМЕРА

Модуль tmr надає доступ до простих таймерів, системного лічильника та часу безвідмовної роботи.

Він призначений для налаштування завдань, що регулярно виникають, операцій тайм-ауту та надання дельт із низькою роздільною здатністю.

Хоча більшість тайм-аутів виражаються в мілісекундах або навіть мікросекундах, точність обмежена, а помилки складення призведуть до досить неточного відліку часу.

tmr.create()	Створює динамічний об'єкт таймера
tmr.delay()	Виконує роботу процесора протягом заданої кількості мікросекунд.
tmr.now()	Повертає системний лічильник, який рахується в мікросекундах.
tmr.softwd()	Надає простий програмний сторожовий таймер, який потрібно переозброїти або вимкнути, перш ніж закінчиться термін його дії, інакше систему буде перезапущено.
tmr.time()	Повертає час безвідмовної роботи системи в секундах.
tmr.wdclr()	Годувати системний сторожовий пес.
tmr.ccount()	Отримати значення регістра CPU CCOUNT, який містить тики ЦП.

ОПИС ФУНКЦІЙ API МОДУЛЯ ТАЙМЕРА

tmr.create()

Створює динамічний об'єкт таймера; таблицю методів дивіться нижче.

Динамічний таймер можна використовувати замість числового ідентифікатора в функціях керування. Також можна керувати об'єктно-орієнтованим способом.

Функції, які підтримуються в об'єкті таймера:

- t:alarm()
- t:interval()
- t:register()
- t:start()
- t:state()
- t:stop()
- t:unregister()

Параметри

немає

Повернення

timer об'єкт

приклад

```
local mytimer = tmr.create()
mytimer:register(5000, tmr.ALARM_SINGLE, function (t) print("expired");
t:unregister() end)
mytimer:start()
```

tmr.delay()

Виконує роботу процесора протягом заданої кількості мікросекунд.

Синтаксис

tmr.delay(us)

Параметри

`us` мікросекунди до busyloop для

Повернення

`nil`

приклад

```
tmr.delay(100)
```

tmr.now()

Повертає системний лічильник, який рахується в мікросекундах. Обмежується 31 бітом, після чого повертається до нуля. .

Синтаксис

`tmr.now()`

Параметри

немає

Повернення

поточне значення системного лічильника

приклад

```
print(tmr.now())
print(tmr.now())
```

tmr.softwd()

Надає простий програмний сторожовий таймер, який потрібно переозброїти або вимкнути, перш ніж закінчиться термін його дії, інакше систему буде перезапущено.

Синтаксис

`tmr.softwd(timeout_s)`

Параметри

`timeout_s` тайм-аут сторожового таймера, у секундах. Щоб вимкнути сторожовий таймер, використовуйте -1 (або будь-яке інше від’ємне значення).

Повернення

`nil`

приклад

```
function on_success_callback()
  tmr.softwd(-1)
  print("Complex task done, soft watchdog disabled!")
end

tmr.softwd(5)
-- go off and attempt to do whatever might need a restart to recover from
complex_stuff_which_might_never_call_the_callback(on_success_callback)
```

tmr.time()

Повертає час безвідмовної роботи системи в секундах. Обмежується 31 бітом, після чого повертається до нуля.

Синтаксис

`tmr.time()`

Параметри

немає

Повернення

час безперебійної роботи системи в секундах, можливо, загорнутий

приклад

```
print("Uptime (probably):", tmr.time())
```

tmr.wdclr()

Системний сторожовий таймер.

Синтаксис

`tmr.wdclr()`

Параметри

немає

Повернення

`nil`

tmr.ccount()

Отримати значення регістра CPU CCOUNT, який містить тики ЦП. Реєстр 32-розрядний.

Перетворення тактів процесора реєстру в нас здійснюється шляхом ділення його на 80 або 160 (CPU80/CPU160), тобто `tmr.ccount() / node.getcpufreq()`.

Реєстраційна арифметика працює без необхідності враховувати перекидання, на відміну від `tmr.now()`. З тієї ж причини, коли CCOUNT має свій 32-й біт, він відображається в Lua як від'ємне число.

Синтаксис

`tmr.ccount()`

Повернення

Поточне значення регістру CCOUNT.

приклад

```
function timeIt(fnc, cnt)
   local function loopIt(f2)
     local t0 = tmr.ccount()
     for i=1,cnt do
       f2()
     end
     local t1 = tmr.ccount()
     return math.ceil((t1-t0)/cnt)
   end
   assert(type(fnc) == "function", "function to test missing")
   cnt = cnt or 1000
   local emptyTime = loopIt(function()end)
   local deltaCPUTicks = math.abs(loopIt(fnc) - emptyTime)
   local deltaUS = math.ceil(deltaCPUTicks/node.getcpufreq())
   return deltaCPUTicks, deltaUS
end
print( timeIt(function() tmr.ccount() end) )
```

Методи об'єкта Timer

tobj:alarm()

Це зручна функція, яка об’єднує `tobj:register()` і `tobj:start()` в один виклик. Ось чому цей метод має ті самі параметри, що й `tobj:register()`. Якщо `tobj:alarm()` викликається на вже запущеному таймері, таймер зупиняється, встановлюються нові параметри та (пере)запускається (подібно до виклику `tobj:start(true)`).

Щоб звільнити ресурси за допомогою цього таймера, коли закінчите використовувати його, зателефонуйте `tobj:unregister()` йому. Для одноразових таймерів це не потрібно, якщо їх не було зупинено до закінчення терміну дії.

Синтаксис

`tobj:alarm(interval_ms, mode, func())`

Параметри

- `interval_ms` інтервал таймера в мілісекундах. Максимальне значення становить 6870947 (1:54:30,947).
- `mode` режим таймера:
  - `tmr.ALARM_SINGLE` одноразова сигналізація (і не потрібно викликати `unregister()`)
  - `tmr.ALARM_SEMI` ручне повторення таймера (виклик `start()` для перезапуску)
  - `tmr.ALARM_AUTO` автоматичне повторення таймера
- `func(timer)` функція зворотного виклику, яка викликається з об’єктом таймера як аргументом

Повернення

`true` якщо таймер був запущений, `false` про помилку

приклад

```
if not tmr.create():alarm(5000, tmr.ALARM_SINGLE, function()
  print("hey there")
end)
then
  print("whoopsie")
end
```

Дивись також

- `tobj:create()`
- `tobj:register()`
- `tobj:start()`
- `tobj:unregister()`

tobj:interval()

Змінює інтервал закінчення зареєстрованого таймера.

Синтаксис

`tobj:interval(interval_ms)`

Параметри

- `interval_ms` новий інтервал таймера в мілісекундах. Максимальне значення становить 6870947 (1:54:30,947).

Повернення

`nil`

приклад

```
mytimer = tmr.create()
mytimer:register(10000, tmr.ALARM_AUTO, function() print("hey there") end)
mytimer:interval(3000) -- actually, 3 seconds is better!
mytimer:start()
```

tobj:register()

Налаштовує таймер і реєструє функцію зворотного виклику для виклику після закінчення терміну дії.

Щоб звільнити ресурси за допомогою цього таймера, коли закінчите використовувати його, зателефонуйте `tobj:unregister()` йому. Для одноразових таймерів це не потрібно, якщо їх не було зупинено до закінчення терміну дії.

Синтаксис

`tobj:register(interval_ms, mode, func())`

Параметри

- `interval_ms` інтервал таймера в мілісекундах. Максимальне значення становить 6870947 (1:54:30,947).
- `mode` режим таймера:
  - `tmr.ALARM_SINGLE` одноразова сигналізація (і не потрібно викликати `tobj:unregister()`)
  - `tmr.ALARM_SEMI` ручне повторення таймера (виклик `tobj:start()` для перезапуску)
  - `tmr.ALARM_AUTO` автоматичне повторення таймера
- `func(timer)` функція зворотного виклику, яка викликається з об’єктом таймера як аргументом

Зауважте, що реєстрація *не* запускає таймер.

Повернення

`nil`

приклад

```
mytimer = tmr.create()
mytimer:register(5000, tmr.ALARM_SINGLE, function() print("hey
there") end)
mytimer:start()
```

Дивись також

- tobj:create()
- tobj:alarm()

tobj:start()

Запускає або перезапускає раніше налаштований таймер. Якщо таймер працює, таймер перезапускається лише тоді, коли restart параметр має значення true. В іншому випадку false повертається повідомлення про помилку.

Синтаксис

tobj:start([restart])

Параметри

- restart необов'язковий логічний параметр, що змушує перезапустити вже запущений таймер

Повернення

true якщо таймер було (пере)запущено, false про помилку

приклад

```
mytimer = tmr.create()
mytimer:register(5000, tmr.ALARM_SINGLE, function() print("hey there")
end)
if not mytimer:start() then print("uh oh") end
```

Дивись також

- tobj:create()
- tobj:register()
- tobj:stop()
- tobj:unregister()

tobj:state()

Перевіряє стан таймера.

Синтаксис

`tobj:state()`

Параметри

Жодного

Повернення

(bool, int) або `nil`

Якщо вказаний таймер зареєстровано, повертає інформацію про те, чи він наразі запущений, і його режим. Якщо таймер не зареєстровано, `nil` повертається.

приклад

```
mytimer = tmr.create()
print(mytimer:state()) -- nil
mytimer:register(5000, tmr.ALARM_SINGLE, function() print("hey there")
end)
running, mode = mytimer:state()
print("running: " .. tostring(running) .. ", mode: " .. mode) -- running:
false, mode: 0
```

tobj:stop()

Зупиняє запущений таймер, але *не* скасовує його реєстрацію. Зупинений таймер можна перезапустити за допомогою `tobj:start()`.

Синтаксис

`tobj:stop()`

Параметри

Жодного

Повернення

`true` якщо таймер було зупинено, `false` про помилку

приклад

```
mytimer = tmr.create()
if not mytimer:stop() then print("timer not stopped, not registered?") end
```

Дивись також

- `tobj:register()`
- `tobj:stop()`
- `tobj:unregister()`

tobj:unregister()

Зупиняє таймер (якщо він запущений) і скасовує реєстрацію відповідного зворотного виклику.

Синтаксис

`tobj:unregister()`

Параметри

Жодного

Повернення

`nil`

Дивись також

`tobj:register()`

МОДУЛЬ FIFO

Цей модуль забезпечує гнучкі загальні FIFO, побудовані навколо таблиць Lua та функцій зворотного виклику. Він спеціально розроблений для роботи з MCU на основі подій і обмеженням пам'яті.

Конструктор

```
fifo = (require "fifo").new()
```

fifo.dequeue()

Синтаксис

`fifo:dequeue(k)`

Отримати елемент із FIFO та передати його функції `k` разом із логічним значенням, яке вказує, чи є це останнім елементом у FIFO. Якщо fifo порожній, `k` він не буде викликаний і fifo перейде в режим «негайного вилучення з черги» (див. нижче).

Якщо припустити `k`, що викликається, зазвичай `k` буде повернуто `nil`, що спричинить `k` видалення елемента з FIFO та перехід черги.

Якщо, однак, `k` повертається не- `nil` значення, це значення замінить елемент на початку FIFO. Це може бути корисно для генераторів, наприклад, які замінюють кілька елементів.

Коли `k` повертає `nil`, він також може повернути логічне значення як другий результат.

Якщо це так `false`, обробка завершується та `fifo:dequeue` повертається. Якщо це так `true`, fifo буде знову просунуто (тобто `fifo:dequeue(k)` буде *викликано хвіст*).

Елементи, для яких `k` повертаються, `nil, true` називаються "фантомними", оскільки вони змушують fifo діяти так, ніби їх там не було.

Фантомні елементи корисні для поведінки, подібної до зворотного виклику, у міру просування fifo: коли `k` бачить фантомний елемент, він знає, що всі попередні записи у fifo були переглянуті, але фантом не обов’язково знає, як згенерувати наступний елемент fifo.

Повернення

`true` якщо черга містила принаймні один нефантомний

запис, `false` інакше.

fifo.queue()

Синтаксис

`fifo:queue(a,k)`

Встановлює елемент `a` у чергу у fifo. Якщо `k` ні, `nil` і fifo знаходиться в режимі «негайного вилучення з черги» (звідки він починається), негайно передайте перший елемент fifo (зазвичай, але не обов’язково, `a`) до `k`, ніби `fifo:dequeue(k)` його було викликано, і вийдіть з режиму «негайного вилучення з черги».

Елементи FIFO

Елементи, що зберігаються у FIFO, є просто цілочисельними індексами самої таблиці FIFO, яка `1` є головою FIFO. Глибина черги для заданого `fifo` – це просто розмір його таблиці, тобто `#fifo`.

Прямий доступ до елементів настійно не рекомендується. Кількість елементів у fifo також навряд чи буде цікавою; зокрема, рішення про порожнечу FIFO слід натомість переписати, щоб використовувати існуючий інтерфейс, якщо це можливо, або можна трохи заглянути в стан негайного вилучення з черги.

Негайне видалення з черги

Поведінка «миттєвого вилучення з черги» може здатися нелогічною, але вона дуже корисна для випадку, коли fifo:dequeue організовуються k наступні виклики fifo:dequeue, скажімо, шляхом планування наступного виклику таймера або надсилання на сокет із on("sent") зворотним викликом, підключеним до fifo:dequeue.

Оскільки fifo переходить у режим «негайного вилучення з черги» лише тоді, коли dequeue його було викликано, а fifo був порожнім під час виклику, а не тоді, коли fifo *стає* порожнім, fifo:queue інколи не буде викликати його, k навіть якщо елемент у черзі a закінчується на початку фіфо.

Це теж дуже корисно: це гарантує, що k не буде викликано в контекстах, де це буде перекривати будь-яку поточну обробку останнього видаленого з черги елемента, що спорожнює fifo.

Статус миттєвого розблокування FIFO видно як _go член, який можна прочитати (навіть якщо зазначені читання ввічливо не рекомендуються, але іноді це зручно знати), але ніколи не слід писати.

Укупорка

Fifo не має спеціальної підтримки для закупорювання (тобто, постановки в чергу кількох елементів, які гарантовано не будуть вилучені з черги до деякого наступного моменту, який називається «відкорковування»). Оскільки часто хочеться закупорити лише тоді, коли FIFO переходить із режиму негайного розблокування, існуючий механізм, як правило, достатньо хороший, щоб забезпечити його легку емуляцію. Хоча типово передавати те саме k для обох :queue і :dequeue, немає нічого, що вимагало б цієї угоди. І тому можна, як

і в fifosock модулі, використовувати :queue k для запису переходу з режиму негайного вилучення з черги на потім, коли ви бажаєте відкоркувати:

```
local corked = false
fifo:queue(e1, function(e) corked = true ; return e end)
  -- e1 is now in the fifo, and corked is true if the fifo has exited
  -- immediate dequeue mode.  e1 will be returned back to the fifo and
  -- so will not be dequqed by the function argument.

  -- We can now queue more elements to the fifo.  These will certainly
  -- queue behind e1.
fifo:queue(e2)

  -- If we should have initiated draining the fifo above, we can do so
now,
  -- instead, having built up a backlog as desired.
if corked then fifo:dequeue(k) end
```

МОДУЛЬ I²C

I²C (I2C, IIC) - послідовна 2-провідна шина для зв'язку з різними пристроями. Також відомий як SMBus або TWI, хоча SMBus має деякі доповнення до протоколу I2C.

Мікросхема ESP8266 не має апаратного I²C, тому модуль використовує програмний драйвер I²C. Його можна налаштувати на будь-яких контактах GPIO, включаючи GPIO16.

Високошвидкісний режим (тактова частота 3,5 МГц) і 10-бітна схема адресації не підтримуються.

Ви повинні викликати `i2c.setup` на дану шину I²C принаймні один раз, перш ніж спілкуватися з будь-яким пристроєм, підключеним до цієї шини, інакше ви отримаєте повідомлення про помилку.

Шина I²C розроблена для роботи в режимі відкритого стоку, тому їй потрібні підтягуючі резистори 1–10 кОм на лініях SDA та SCL. Хоча багато периферійних модулів мають вбудовані підтягуючі резистори і працюватимуть без додаткових зовнішніх резисторів.

Підказка щодо використання багатьох ідентичних пристроїв з однаковою адресою: багато пристроїв дозволяють вибирати між 2 адресами I²C через пін або припаяний резистор 0 Ом.

Якщо зміна адреси неможлива або вам потрібно використовувати більше 2 подібних пристроїв, ви можете використовувати різні шини I²C. Ініціалізуйте їх один раз, зателефонувавши `i2c.setup` з різними номерами шин і PIN-кодами, а потім зверніться до кожного пристрою за ідентифікатором шини та адресою пристрою.

Виводи SCL мають бути різними, SDA можна спільно використовувати на одному виводі.

Зверніть увагу, що раніше багато драйверів і модулів NodeMCU припускали, що доступна лише одна шина I²C з ідентифікатором 0, тому завжди безпечніше починати з ідентифікатора 0 як першої шини у вашому коді.

Якщо ваші функції драйвера пристрою не мають ідентифікатора шини I²C як вхідний параметр і/або не побудовані за принципами Lua OOP, то, швидше за все, пристрій буде доступним лише через ідентифікатор шини 0 і має бути підключений до його контактів.

`#define I2C_MASTER_OLD_VERSION` Щоб увімкнути новий рядок коментарів драйвера `user_config.h`

Щоб увімкнути підтримку GPIO16 (D0), розкоментуйте `#define I2C_MASTER_GPIO16_ENABLED` рядок `user_config.h`

GPIO16 не підтримує режим відкритого стоку і працює в режимі push-pull. Це може призвести до помилок зв'язку, коли підлеглий пристрій намагається розтягнути тактовий сигнал SCL, але не може утримувати лінію SCL на низькому рівні. Якщо це станеться, спробуйте встановити нижчу швидкість I²C.

i2c.address()	Налаштувати адресу I²C і режим читання/запису для наступної передачі.
i2c.read()	Читання даних для змінної кількості байтів.
i2c.setup()	Ініціалізувати шину I²C вибраним номером шини, контактами та швидкістю.
i2c.start()	Надіслати умову запуску I²C.
i2c.stop()	Надіслати умову зупинки I²C.
i2c.write()	Запис даних на шину I²C.

ОПИС ФУНКЦІЙ API МОДУЛЯ I2C

i2c.address()

Налаштувати адресу I²C і режим читання/запису для наступної передачі. На шині I²C кожен пристрій адресується 7-бітним числом. Адресу конкретного пристрою можна знайти в його технічному паспорті.

Синтаксис

`i2c.address(id, device_addr, direction)`

Параметри

- `id` номер автобуса
- `device_addr` 7-розрядна адреса пристрою. Пам'ятайте, що в I²C `device_addr` позначає старші 7 бітів, за якими йде один `direction` біт. Іноді адреса пристрою рекламується як 8-бітове значення, тоді розділити його на 2, щоб отримати 7-бітне значення.
- `direction` `i2c.TRANSMITTER` для режиму запису , `i2c. RECEIVER` для режиму читання

Повернення

`true` якщо підтвердження отримано, `false` якщо підтвердження не отримано.

Дивись також

i2c.read()

i2c.read()

Читання даних для змінної кількості байтів.

Синтаксис

`i2c.read(id, len)`

Параметри

- `id` номер автобуса
- `len` кількість байтів даних

Повернення

`string` отриманих даних

приклад

```
id  = 0
sda = 1
scl = 2

-- initialize i2c, set pin 1 as sda, set pin 2 as scl
i2c.setup(id, sda, scl, i2c.FAST)

-- user defined function: read 1 byte of data from device
function read_reg(id, dev_addr, reg_addr)
    i2c.start(id)
    i2c.address(id, dev_addr, i2c.TRANSMITTER)
    i2c.write(id, reg_addr)
    i2c.stop(id)
    i2c.start(id)
    i2c.address(id, dev_addr, i2c.RECEIVER)
    c = i2c.read(id, 1)
    i2c.stop(id)
    return c
end

-- get content of register 0xAA of device 0x77
reg = read_reg(id, 0x77, 0xAA)
print(string.byte(reg))
```

Дивись також

i2c.write()

i2c.setup()

Ініціалізувати шину I²C вибраним номером шини, контактами та швидкістю.

Синтаксис

`i2c.setup(id, pinSDA, pinSCL, speed)`

Параметри

- `id` 0~9, номер автобуса
- `pinSDA` 1~12, індекс IO
- `pinSCL` 0~12, індекс IO
- `speed` `i2c.SLOW` (100 кГц), `i2c.FAST` (400 кГц), `i2c.FASTPLUS` (1 МГц) або будь-яка тактова частота в діапазоні 25000-1000000 Гц. Режим FASTPLUS забезпечує тактову частоту I2C 600 кГц при частоті ЦП 80 МГц за замовчуванням. Щоб отримати тактову частоту I2C 1 МГц, змініть частоту процесора на 160 МГц за допомогою функції `node.setcpufreq(node.CPU160MHZ)`.

Повернення

`speed` вибрана швидкість, `0` якщо помилка ініціалізації шини.

приклад

```
i2c0 = {
  id  = 0,
  sda = 1,
  scl = 0,
  speed = i2c.FAST
}
i2c1 = {
  id  = 1,
  sda = 1,
  scl = 2,
  speed = i2c.FASTPLUS
}
-- initialize i2c bus 0
i2c0.speed = i2c.setup(i2c0.id, i2c0.sda, i2c0.scl, i2c0.speed)
-- initialize i2c bus 1 with shared SDA on pin 1
node.setcpufreq(node.CPU160MHZ) -- to support FASTPLUS speed
i2c1.speed = i2c.setup(i2c1.id, i2c1.sda, i2c1.scl, i2c1.speed)
print("i2c bus 0 speed: ", i2c0.speed, "i2c bus 1 speed: ", i2c1.speed)
```

Дивись також

i2c.read()

i2c.start()

Надіслати умову запуску I²C.

Синтаксис

`i2c.start(id)`

Параметри

`id` номер автобуса

Повернення

`nil`

Дивись також

i2c.read()

i2c.stop()

Надіслати умову зупинки I²C.

Синтаксис

`i2c.stop(id)`

Параметри

`id` номер автобуса

Повернення

`nil`

Дивись також

i2c.read()

i2c.write()

Запис даних на шину I²C. Елементами даних можуть бути кілька чисел, рядків або таблиць Lua.

Синтаксис

`i2c.write(id, data1[, data2[, ..., datan]])`

Параметри

- `id` номер автобуса
- `data` дані можуть бути числами, рядками або таблицею Lua.

Повернення

`number` кількість записаних байтів

приклад

```
id  = 0
sda = 1
scl = 2

-- initialize i2c, set pin 1 as sda, set pin 2 as scl
i2c.setup(id, sda, scl, i2c.FAST)

-- user defined function: write some data to device
-- with address dev_addr starting from reg_addr
function write_reg(id, dev_addr, reg_addr, data)
    i2c.start(id)
    i2c.address(id, dev_addr, i2c.TRANSMITTER)
    i2c.write(id, reg_addr)
    c = i2c.write(id, data)
    i2c.stop(id)
    return c
end
-- set register with address 0x45 of device 0x77 with value 1
count = write_reg(id, 0x77, 0x45, 1)
print(count, " bytes written")

-- write text into i2c EEPROM starting with memory address 0
count = write_reg(id, 0x50, 0, "Sample")
print(count, " bytes written")
```

Дивись також

i2c.read()

МОДУЛЬ SPI

Усі транзакції для надсилання та отримання мають старший біт першим і найменш значущий біт останнім. **Примітка**

Апаратне забезпечення ESP забезпечує дві шини SPI з ідентифікаторами 0 та 1, які відображаються на контакти, які зазвичай позначаються як SPI та HSPI.

Сигнали HSPI фіксуються до таких індексів вводу-виводу та контактів GPIO:

Сигнал	Індекс IO	Пін ESP8266
HSPI CLK	5	GPIO14
HSPI / CS	8	GPIO15
HSPI MOSI	7	GPIO13
HSPI MISO	6	GPIO12

Функції високого рівня	Функції високого рівня забезпечують API надсилання та отримання для напівдуплексного та повнодуплексного режимів.
spi.recv()	Отримувати дані від SPI.
spi.send()	Надсилати дані через SPI в напівдуплексному режимі.
spi.setup()	Налаштувати конфігурацію SPI.
spi.set_clock_div()	Встановіть дільник тактової частоти SPI.
Апаратні функції низького рівня	Функції низького рівня забезпечують апаратно-орієнтований API для прикладних сценаріїв, які потребують виконання складніших транзакцій SPI.
spi.get_miso()	Витяг елементів даних із буфера MISO після spi.
spi.set_mosi()	Вставити елементи даних у буфер MOSI для spi.

ОПИС ФУНКЦІЙ API МОДУЛЯ SPI

Функції високого рівня забезпечують API надсилання та отримання для напівдуплексного та повнодуплексного режимів. Відправлені та отримані елементи даних обмежені довжиною від 1 до 32 бітів, і кожен елемент даних оточений неактивним (H)SPI CS.

spi.recv()

Отримувати дані від SPI.

Синтаксис

`spi.recv(id, size[, default_data])`

Параметри

- `id` Ідентифікаційний номер SPI: 0 для SPI, 1 для HSPI
- `size` кількість елементів даних для читання
- `default_data` дані за замовчуванням надсилаються на MOSI (все-1, якщо опущено)

Повернення

Рядок, що містить байти, зчитані з SPI.

Дивись також

spi.send()

spi.send()

Надіслати дані через SPI в напівдуплексному режимі. Надсилайте й отримуйте дані в повнодуплексному режимі.

Синтаксис

Напівдуплекс:

`wrote = spi.send(id, data1[, data2[, ..., datan]])`

Повнодуплексний:

`wrote[, rdata1[, ..., rdatan]] = spi.send(id, data1[, data2[, ..., datan]])`

Параметри

- `id` Ідентифікаційний номер SPI: 0 для SPI, 1 для HSPI
- `data` дані можуть бути рядком, таблицею або цілим числом.

  Кожен елемент даних розглядається з `databits` кількістю бітів.

Повернення

- `wrote` кількість записаних байтів
- `rdata` отримані дані, якщо налаштовано `spi.FULLDUPLEX`

  такий самий тип даних, як і відповідний параметр даних.

Дивись також

- spi.setup()
- spi.recv()

spi.setup()

Налаштувати конфігурацію SPI. Виклик `spi.setup()` направлятиме сигнали HSPI на відповідні контакти, перекриваючи попередню конфігурацію та керування модулем `gpio`.

Можна повернути будь-який пін назад до керування gpio, якщо його функції HSPI не потрібні, просто встановіть `gpio.mode()` для нього потрібний.

Це рекомендовано особливо для функції виводу HSPI /CS у випадку, якщо SPI slave-select керується іншим виводом `gpio.write()` - інакше механізм SPI перемкне вивід 8.

Синтаксис

`spi.setup(id, mode, cpol, cpha, databits, clock_div[, duplex_mode])`

Параметри

- id Ідентифікаційний номер SPI: 0 для SPI, 1 для HSPI
- mode виберіть головний або підлеглий режим
  - spi.MASTER
  - spi.SLAVE - **наразі не підтримується**
- cpol вибір полярності годинника
  - spi.CPOL_LOW
  - spi.CPOL_HIGH
- cpha вибір фази годинника
  - spi.CPHA_LOW
  - spi.CPHA_HIGH
- databits кількість бітів на елемент даних 1 - 32
- clock_div Дільник тактової частоти SPI, f(SPI) = 80 МГц / clock_div, 1 .. n (0 за замовчуванням дільник 8)
- duplex_mode дуплексний режим
  - spi.HALFDUPLEX (за умовчанням, якщо опущено)
  - spi.FULLDUPLEX

Повернення

Номер 1

приклад

```
spi.setup(1, spi.MASTER, spi.CPOL_LOW, spi.CPHA_LOW, 8, 8)
-- we won't be using the HSPI /CS line, so disable it again
gpio.mode(8, gpio.INPUT, gpio.PULLUP)
```

spi.set_clock_div()

Встановіть дільник тактової частоти SPI.

Синтаксис

old_div = spi.set_clock_div(id, clock_div)

Параметри

- id Ідентифікаційний номер SPI: 0 для SPI, 1 для HSPI
- clock_div Дільник тактової частоти SPI, f(SPI) = 80 МГц / clock_div, 1 .. n

приклад

```
old_div = spi.set_clock_div(1, 84) --drop to slow clock for slow device
spi.send(1, 0x0B, 0xFF)
spi.set_clock_div(1, old_div)
```

Апаратні функції низького рівня

Функції низького рівня забезпечують апаратно-орієнтований API для прикладних сценаріїв, які потребують виконання складніших транзакцій SPI. Модель програмування побудована навколо апаратних буферів надсилання та отримання, а транзакції SPI ініціюються з повним контролем над апаратними функціями.

spi.get_miso()

Витягти елементи даних із буфера MISO після spi.transaction().

Синтаксис

data1[, data2[, ..., datan]] = spi.get_miso(id, offset, bitlen, num)

string = spi.get_miso(id, num)

Параметри

- `id` Ідентифікаційний номер SPI: 0 для SPI, 1 для HSPI
- `offset` бітовий зсув у буфер MISO для першого елемента даних
- `bitlen` довжина в бітах одного елемента даних
- `num` кількість елементів даних для отримання

Повернення

`num` елементи даних або `string`

Дивись також

spi.transaction()

spi.set_mosi()

Надіслати елементи даних у буфер MOSI для `spi.transaction()`.

Синтаксис

```
spi.set_mosi(id, offset, bitlen, data1[, data2[, ..., datan]])
```

```
spi.set_mosi(id, string)
```

Параметри

- `id` Ідентифікаційний номер SPI: 0 для SPI, 1 для HSPI
- `offset` бітове зміщення в буфер MOSI для вставки data1 і наступних елементів
- `bitlen` довжина даних1, даних2, ...
- `data` елементи даних, де `bitlen` для транзакції враховується кількість бітів.
- `string` відправити дані для копіювання в буфер MOSI зі зміщенням 0, довжиною бітів 8

Повернення

`nil`

Дивись також

spi.transaction()

spi.transaction()

Розпочніть транзакцію SPI, яка складається з 5 етапів:

1. Команда
2. Адреса
3. MOSI
4. Шаблон
5. MISO

Синтаксис

`spi.transaction(id, cmd_bitlen, cmd_data, addr_bitlen, addr_data, mosi_bitlen, dummy_bitlen, miso_bitlen)`

Параметри

- `id` Ідентифікаційний номер SPI: 0 для SPI, 1 для HSPI
- `cmd_bitlen` розрядність фази команди (0 - 16)
- `cmd_data` дані для командної фази
- `addr_bitlen` довжина біта для фази адреси (0 - 32)
- `addr_data` дані для фази адреси
- `mosi_bitlen` розрядність фази MOSI (0 - 512)
- `dummy_bitlen` розрядність фіктивної фази (0 - 256)
- `miso_bitlen` довжина біта фази MISO (0 - 512) для напівдуплексу.

  Повнодуплексний режим активовано з від’ємним значенням.

Повернення

`nil`

КРИПТОГРАФІЧНИЙ МОДУЛЬ

Криптомодулі забезпечують різні функції для роботи з криптоалгоритмами.

Підтримуються наступні алгоритми/режими шифрування/дешифрування: - `"AES-ECB"` для 128-бітного AES у режимі ECB - `"AES-CBC"` для 128-бітного AES у режимі CBC

Підтримуються наступні алгоритми хешування: - MD5 - SHA1 - SHA256, SHA384, SHA512 (якщо не вимкнено в `app/include/user_config.h`)

crypto.encrypt()	Шифрує рядки Lua.
crypto.decrypt()	Розшифровує раніше зашифровані дані.
crypto.fhash()	Обчисліть криптографічний хеш файлу аа.
crypto.hash()	Обчисліть криптографічний хеш рядка Lua.
crypto.new_hash()	Створіть дайджест/хеш-об’єкт, до якого можна додати будь-яку кількість рядків.
crypto.hmac()	Обчисліть підпис HMAC (хешований код автентифікації повідомлення) для рядка Lua.
crypto.new_hmac()	Створіть об’єкт hmac, до якого можна додати будь-яку кількість рядків.
crypto.mask()	Застосовує маску XOR до рядка Lua.

ОПИС ФУНКЦІЙ API МОДУЛЯ CRYPTO

crypto.encrypt()

Шифрує рядки Lua.

Синтаксис

```
crypto.encrypt(algo, key, plain [, iv])
```

Параметри

- `algo` назва підтримуваного алгоритму шифрування для використання
- `key` ключ шифрування у вигляді рядка; для шифрування AES

  це *ПОВИННО* мати довжину 16 байт
- `plain` рядок для шифрування; він буде автоматично доповнений

  нулями до 16-байтової межі, якщо необхідно
- `iv` вектор ініціалізації, якщо використовується AES-CBC; за

  замовчуванням - усі нулі, якщо не задано

Повернення

Зашифровані дані у вигляді двійкового рядка. Для AES це завжди кратне 16 байтам.

приклад

```
print(encoder.toHex(crypto.encrypt("AES-ECB", "1234567890abcdef", "Hi, I'm
secret!")))
```

Дивись також

- `crypto.decrypt()`

crypto.decrypt()

Розшифровує раніше зашифровані дані.

Синтаксис

`crypto.decrypt(algo, key, cipher [, iv])`

Параметри

- `algo` назва підтримуваного алгоритму шифрування для використання
- `key` ключ шифрування у вигляді рядка; для шифрування AES

  це *ПОВИННО* мати довжину 16 байт
- `cipher` зашифрований текст для розшифровки (як отримано

  з `crypto.encrypt()`)

- `iv` вектор ініціалізації, якщо використовується AES-CBC; за замовчуванням - усі нулі, якщо не задано

Повернення

Розшифрований рядок.

Зауважте, що розшифрований рядок може містити зайві нульові байти доповнення в кінці. Одним із способів видалення такого доповнення є використання `:match("(.-)%z*$")` на розшифрованому рядку.

Необхідно бути особливо обережним, якщо працюєте з двійковими даними, у цьому випадку реальна довжина, ймовірно, повинна бути закодована разом з даними, і тоді її `:sub(1, n)` можна використовувати для видалення заповнення.

приклад

```
key = "1234567890abcdef"
cipher = crypto.encrypt("AES-ECB", key, "Hi, I'm secret!")
print(encoder.toHex(cipher))
print(crypto.decrypt("AES-ECB", key, cipher))
```

Дивись також

- `crypto.encrypt()`

crypto.fhash()

Обчисліть криптографічний хеш файлу аа.

Синтаксис

`hash = crypto.fhash(algo, filename)`

Параметри

- `algo` використовуваний хеш-алгоритм, рядок без урахування регістру
- `filename` шлях до файлу для хешування

Повернення

Двійковий рядок, що містить дайджест повідомлення. Щоб отримати текстову версію (шістнадцяткові символи ASCII), використовуйте `encoder.toHex()`.

приклад

```
print(encoder.toHex(crypto.fhash("sha1","myfile.lua")))
```

crypto.hash()

Обчисліть криптографічний хеш рядка Lua.

Синтаксис

`hash = crypto.hash(algo, str)`

Параметри

`algo` алгоритм хешування для використання, рядок без урахування регістру, `str` для хешування вмісту

Повернення

Двійковий рядок, що містить дайджест повідомлення. Щоб отримати текстову версію (шістнадцяткові символи ASCII), використовуйте `encoder.toHex()`.

приклад

```
print(encoder.toHex(crypto.hash("sha1","abc")))
```

crypto.new_hash()

Створіть дайджест/хеш-об’єкт, до якого можна додати будь-яку кількість рядків. Об'єкт має `update` і `finalize` функції.

Синтаксис

`hashobj = crypto.new_hash(algo)`

Параметри

`algo` використовуваний хеш-алгоритм, рядок без урахування регістру

Повернення

Об’єкт даних користувача з доступними update функціями finalize.

приклад

```
hashobj = crypto.new_hash("SHA1")
hashobj:update("FirstString")
hashobj:update("SecondString")
digest = hashobj:finalize()
print(encoder.toHex(digest))
```

crypto.hmac()

Обчисліть підпис HMAC (хешований код автентифікації повідомлення) для рядка Lua.

Синтаксис

signature = crypto.hmac(algo, str, key)

Параметри

- algo використовуваний хеш-алгоритм, рядок без урахування регістру
- str дані для обчислення хешу
- key ключ для підпису може бути двійковим рядком

Повернення

Двійковий рядок, що містить підпис HMAC.

Використовуйте encoder.toHex() для отримання текстової версії.

приклад

```
print(encoder.toHex(crypto.hmac("sha1","abc","mysecret")))
```

crypto.new_hmac()

Створіть об’єкт hmac, до якого можна додати будь-яку кількість рядків.

Об'єкт має update і finalize функції.

Синтаксис

hmacobj = crypto.new_hmac(algo, key)

Параметри

- algo використовуваний хеш-алгоритм, рядок без урахування регістру
- key ключ для використання (може бути двійковий рядок)

Повернення

Об’єкт даних користувача з доступними update функціями finalize.

приклад

```
hmacobj = crypto.new_hmac("SHA1", "s3kr3t")
hmacobj:update("FirstString")
hmacobj:update("SecondString")
digest = hmacobj:finalize()
print(encoder.toHex(digest))
```

crypto.mask()

Застосовує маску XOR до рядка Lua. Зауважте, що це неналежний криптографічний механізм, але деякі протоколи все ж можуть його використовувати.

Синтаксис

crypto.mask(message, mask)

Параметри

- message повідомлення для маскування
- mask маска, яку потрібно застосувати, повторюється, якщо вона коротша за повідомлення

Повернення

Замасковане повідомлення у вигляді двійкового рядка.

Використовуйте encoder.toHex(), щоб отримати його текстове представлення.

приклад

```
print(encoder.toHex(crypto.mask("some message to obscure","X0Y7")))
```

МОДУЛЬ 1-WIRE

Цей модуль забезпечує функції для роботи з системою комунікаційних шин пристроїв 1-Wire .

ow.check_crc16()	Обчислює CRC16 1-Wire і порівнює його з отриманим CRC.
ow.crc16()	Обчислює 16-бітний CRC Dallas Semiconductor.
ow.crc8()	Обчислює 8-розрядний CRC компанії Dallas Semiconductor, який використовується в ПЗУ та регістрах блокнота.
ow.depower()	Зупиняє примусову подачу живлення на шині.
ow.read()	Читає байт.
ow.read_bytes()	Читає кілька байтів.
ow.reset()	Виконує цикл скидання 1-Wire.
ow.reset_search()	Очищає стан пошуку, щоб почати знову з початку.
ow.search()	Шукає наступний пристрій.
ow.select()	Видає команду вибору 1-Wire rom.
ow.setup()	Встановлює пін у режимі onewire.
ow.skip()	Видає команду 1-Wire rom skip, щоб звернутись до всіх на шині.
ow.target_search()	Налаштовує пошук для пошуку типу пристрою family_code.
ow.write()	Записує байт.
ow.write_bytes()	Записує кілька байтів.
ow.set_timings()	Налаштуйте різні параметри синхронізації бітів.

ОПИС ФУНКЦІЙ API МОДУЛЯ 1-Wire

ow.check_crc16()

Обчислює CRC16 1-Wire і порівнює його з отриманим CRC.

Синтаксис

`ow.check_crc16(buf, inverted_crc0, inverted_crc1[, crc])`

Параметри

- `buf` рядкове значення, дані для обчислення, контрольна сума в рядку
- `inverted_crc0` LSB отриманого CRC
- `inverted_crc1` MSB отриманого CRC
- `crc` Початкове значення CRC (необов'язково)

Повернення

true, якщо CRC збігається, false в іншому випадку

ow.crc16()

Обчислює 16-бітний CRC Dallas Semiconductor. Це потрібно для перевірки цілісності даних, отриманих від багатьох пристроїв 1-Wire. CRC передається побітово інвертованим.

Залежно від порядку байтів вашого процесора, двійкове представлення двобайтового значення, що повертається, може мати інший порядок байтів, ніж два байти, які ви отримуєте від 1-Wire.

Синтаксис

`ow.crc16(buf[, crc])`

Параметри

- `buf` рядкове значення, дані для обчислення, контрольна сума в рядку
- `crc` Початкове значення CRC (необов'язково)

Повернення

CRC16 за визначенням Dallas Semiconductor

ow.crc8()

Обчислює 8-розрядний CRC компанії Dallas Semiconductor, який використовується в ПЗУ та регістрах блокнота.

Синтаксис

`ow.crc8(buf)`

Параметри

`buf` рядкове значення, дані для обчислення, контрольна сума в рядку

Повернення

Результат CRC як байт

ow.depower()

Зупиняє примусову подачу живлення на шину.

Синтаксис

`ow.depower(pin)`

Параметри

`pin` 1~12, індекс введення/виведення

Повернення

`nil`

Дивись також

- ow.write()
- ow.write_bytes()

ow.read()

Читає байт.

Синтаксис

`ow.read(pin)`

Параметри

`pin` 1~12, індекс введення/виведення

Повернення

байт, прочитаний з підлеглого пристрою

ow.read_bytes()

Читає кілька байтів.

Синтаксис

`ow.read_bytes(pin, size)`

Параметри

- `pin` 1~12, індекс введення/виведення
- `size` кількість байтів для читання з підлеглого пристрою (до 256)

Повернення

`string` байтів, зчитаних із підлеглого пристрою

ow.reset()

Виконує цикл скидання 1-Wire.

Синтаксис

`ow.reset(pin)`

Параметри

`pin` 1~12, індекс введення/виведення

Повернення

- `1` якщо пристрій відповідає імпульсом присутності
- `0` якщо немає пристрою або шина замкнута або іншим чином утримується на низькому рівні понад 250 мкс

ow.reset_search()

Очищає стан пошуку, щоб почати знову з початку.

Синтаксис

`ow.reset_search(pin)`

Параметри

`pin` 1~12, індекс введення/виведення

Повернення

`nil`

ow.search()

Шукає наступний пристрій.

Синтаксис

`ow.search(pin, [alarm_search])`

Параметри

- `pin` 1~12, індекс введення/виведення
- `alarm_search` 1 / 0, якщо 0, виконується звичайний пошук 0xF0 (за замовчуванням, якщо параметр відсутній), якщо 1, виконується ПОШУК 0xEC ALARM SEARCH.

Повернення

`rom_code` рядок довжиною 8 у разі успіху. Він містить rom-код підлеглого пристрою. Повертає `nil`, якщо пошук не вдався.

Дивись також

ow.target_search()

ow.select()

Видає команду вибору 1-Wire rom. Обов’язково зробіть `ow.reset(pin)` перше.

Синтаксис

`ow.select(pin, rom)`

Параметри

- `pin` 1~12, індекс введення/виведення
- `rom` рядкове значення, len 8, rom код підпорядкованого пристрою

Повернення

`nil`

приклад

```
-- 18b20 Example
-- 18b20 Example
pin = 3
ow.setup(pin)
addr = ow.reset_search(pin)
addr = ow.search(pin)

if addr == nil then
  print("No device detected.")
else
  print(addr:byte(1,8))
  local crc = ow.crc8(string.sub(addr,1,7))
  if crc == addr:byte(8) then
    if (addr:byte(1) == 0x10) or (addr:byte(1) == 0x28) then
      print("Device is a DS18S20 family device.")
      tmr.create():alarm(1000, tmr.ALARM_AUTO, function()
          ow.reset(pin)
          ow.select(pin, addr)
          ow.write(pin, 0x44, 1) -- convert T command
          tmr.create():alarm(750, tmr.ALARM_SINGLE, function()
              ow.reset(pin)
              ow.select(pin, addr)
              ow.write(pin,0xBE,1) -- read scratchpad command
              local data = ow.read_bytes(pin, 9)
              print(data:byte(1,9))
              local crc = ow.crc8(string.sub(data,1,8))
              print("CRC="..crc)
              if crc == data:byte(9) then
                 local t = (data:byte(1) + data:byte(2) * 256) * 625
                 local sgn = t<0 and -1 or 1
                 local tA = sgn*t
                 local t1 = math.floor(tA / 10000)
                 local t2 = tA % 10000
                 print("Temperature="..(sgn<0 and "-" or
"")..t1.."."..t2.." Centigrade")
              end
          end)
      end)
    else
```

```
      print("Device family is not recognized.")
    end
  else
    print("CRC is not valid!")
  end
end
```

Дивись також

ow.reset()

ow.setup()

Встановлює пін у режимі onewire.

Синтаксис

`ow.setup(pin)`

Параметри

`pin` 1~12, індекс введення/виведення

Повернення

`nil`

ow.skip()

Видає команду 1-Wire rom skip, щоб звернутись до всіх на шині.

Синтаксис

`ow.skip(pin)`

Параметри

`pin` 1~12, індекс введення/виведення

Повернення

`nil`

ow.target_search()

Встановлює пошук типу пристрою `family_code`. Сам пошук має бути розпочато наступним викликом `ow.search()`.

Синтаксис

`ow.target_search(pin, family_code)`

Параметри

- `pin` 1~12, індекс введення/виведення
- `family_code` байт для сімейного коду

Повернення

`nil`

Дивись також

ow.search()

ow.write()

Записує байт. Якщо `power` дорівнює 1, то дріт тримається високо на кінці для пристроїв з паразитним живленням.

Синтаксис

`ow.write(pin, v, power)`

Параметри

- `pin` 1~12, індекс введення/виведення
- `v` байт для запису на підлеглий пристрій
- `power` 1 для проводу, що тримається високо для пристроїв з паразитним живленням

Повернення

`nil`

Дивись також

ow.depower()

ow.write_bytes()

Записує кілька байтів. Якщо `power` дорівнює 1, то дріт тримається високо на кінці для пристроїв з паразитним живленням.

Синтаксис

`ow.write_bytes(pin, buf, power)`

Параметри

- `pin` 1~12, індекс IO
- `buf` рядок для запису на підлеглий пристрій
- `power` 1 для проводу, що тримається високо для пристроїв з паразитним живленням

Повернення

`nil`

Дивись також

ow.depower()

ow.set_timings()

Встановити різні параметри синхронізації бітів. Деякі повільні користувальницькі пристрої можуть не ідеально працювати з NodeMCU як 1-wire master.

Оскільки модуль NodeMCU ow змінює бітовий протокол 1-wire, можна трохи налаштувати таймінги.

Синтаксис

`ow.set_timings(reset_tx, reset_wait, reset_rx, w1_low, w1_high, w0_low, w0_high, r_low, r_wait, r_delay)`

Параметри

Кожен параметр визначає кількість мікросекунд для затримки на різних етапах процесу 1-провідного біт-бінгінгу. Нульове значення залишить значення без змін.

- `reset_tx` під час скидання низький рівень шини (за замовчуванням 480)
- `reset_wait` чекати імпульсу присутності після скидання (за замовчуванням 70)
- `reset_rx` затримка після перевірки імпульсу присутності (за замовчуванням 410)
- `w1_low` низький рівень шини під час запису 1 слота (за замовчуванням 5)
- `w1_high` залишити шину високою під час запису 1 слота (за замовчуванням 52)
- `w0_low` низький рівень шини під час запису 1 слота (за замовчуванням 65)
- `w0_high` залишити шину високою під час запису 1 слота (за замовчуванням 5)
- `r_low` низький рівень шини під час слота читання (за замовчуванням 5)
- `r_wait` очікування перед читанням рівня шини під час слота читання (за замовчуванням 8)
- `r_delay` затримка після читання рівня шини (за замовчуванням 52)

Повернення

`nil`

приклад

```
-- Give 300uS longer read/write slots for slow MCU based 1-wire slave
ow.set_timings(
    nil, -- reset_tx
    nil, -- reset_wait
    nil, -- reset_rx
    nil, -- w1_low
    352, -- w1_high
    nil, -- w0_low
    305, -- w0_high
    nil, -- r_low
    nil, -- r_wait
    352  -- r_delay
)
```

МОДУЛЬ GPIO

Цей модуль надає доступ до підсистеми GPIO (введення/виведення загального призначення).

Увесь доступ базується на номері індексу вводу/виводу на комплектах розробників NodeMCU, а не на внутрішньому контакті GPIO. Наприклад, контакт D0 на комплекті для розробників зіставлено з внутрішнім контактом 16 GPIO.

Індекс IO	Пін ESP8266	Індекс IO	Пін ESP8266
0 [*]	GPIO16	7	GPIO13
1	GPIO5	8	GPIO15
2	GPIO4	9	GPIO3
3	GPIO0	10	GPIO1
4	GPIO2	11	GPIO9
5	GPIO14	12	GPIO10
6	GPIO12		

** [*] D0(GPIO16) можна використовувати лише як читання/запис gpio. Немає підтримки open-drain/interrupt/pwm/i2c/ow. **

gpio.mode()	Ініціалізація контакту в режимі GPIO, встановлення напрямку входу/виходу контакту та додаткове внутрішнє слабке підтягування.
gpio.read()	Зчитування цифрове значення контакту GPIO.
gpio.serout()	Серіалізувати вихід на основі послідовності часу затримки в мкс.

gpio.trig()	Встановіть або очистіть функцію зворотного виклику для запуску при перериванні для PIN-коду.
gpio.write()	Встановіть значення цифрового контакту GPIO.
gpio.pulse	Це охоплює набір API, які дозволяють генерувати серії імпульсів із точним синхронізацією на кількох контактах.
gpio.pulse.build	Це створює gpio.
gpio.pulse:старт	Це запускає операції виведення.
gpio.pulse:getstate	Це повертає поточний стан.
gpio.pulse:stop	Це призупиняє операцію виведення в майбутньому.
gpio.pulse:скасувати	Це негайно зупиняє операцію виведення.
gpio.pulse:adjust	Це додає (або віднімає) час, який використовуватиметься у випадку мінімальної/максимальної затримки.
gpio.pulse:оновлення	Це може змінити вміст певного кроку в програмі виводу.

ОПИС ФУНКЦІЙ API МОДУЛЯ GPIO

gpio.mode()

Ініціалізація контакту в режимі GPIO, встановлення напрямку входу/виходу контакту та додаткове внутрішнє слабке підтягування.

Синтаксис

```
gpio.mode(pin, mode [, pullup])
```

Параметри

- `pin` пін для налаштування, індекс вводу-виведення
- `mode` один із gpio.OUTPUT, gpio.OPENDRAIN, gpio.INPUT або gpio.INT (режим переривання)

- pullup gpio.PULLUP вмикає слабкий підтягуючий резистор; за замовчуванням gpio.FLOAT

Повернення

nil

приклад

```
gpio.mode(0, gpio.OUTPUT)
```

Дивись також

- gpio.read()
- gpio.write()

gpio.read()

Зчитати цифрове значення контакту GPIO.

Синтаксис

gpio.read(pin)

Параметри

pin pin для читання, індекс IO

Повернення

число, 0 = низький, 1 = високий

приклад

```
-- read value of gpio 0.
gpio.read(0)
```

Дивись також

gpio.mode()

gpio.serout()

Серіалізує вихід на основі послідовності часу затримки в мкс. Після кожної затримки пін перемикається. Після останнього циклу та останньої затримки пін не перемикається.

Функція працює в двох режимах: * синхронний - для роздільної здатності менше 50 мкс, обмежено макс. загальна тривалість, * асинхронна – синхронна робота з меншою деталізацією, але практично необмеженою тривалістю.

Чи обрано асинхронний режим, визначається наявністю параметра `callback`.

Якщо присутня та має тип функції, функція стає асинхронною, а функція зворотного виклику викликається після завершення послідовності.

Якщо параметр є числовим, функція все ще працює асинхронно, але зворотний виклик не викликається після завершення.

Для асинхронної версії мінімальний час затримки не повинен бути меншим за 50 мкс, а максимальний час затримки становить 0x7fffff мкс (~8,3 секунди).

У цьому режимі функція не блокує стек і повертається безпосередньо перед завершенням вихідної послідовності.

Режим апаратного таймера `FRC1_SOURCE` використовується для зміни станів.

Оскільки існує лише один апаратний таймер, існують обмеження щодо того, які модулі можна використовувати одночасно.

Якщо таймер уже використовується, виникне повідомлення про помилку.

Невиконання цього може призвести до проблем з Wi-Fi або прямих збоїв/перезавантажень.

Коротше кажучи, це означає, що сума всіх часів затримки, помножена на кількість циклів, не повинна перевищувати 15 мс.

Синтаксис

```
gpio.serout(pin, start_level, delay_times [, cycle_num[, callback]])
```

Параметри

- pin pin для використання, індекс IO
- start_level рівень для початку або gpio.HIGH або gpio.LOW
- delay_times масив часів затримки в мкс між кожним перемиканням виводу gpio.
- cycle_num необов'язкову кількість разів для проходження послідовності. (за замовчуванням 1)
- callback необов'язкова функція або номер зворотного виклику, якщо вона присутня, функція негайно повертається та стає асинхронною.

Повернення

nil

приклад

```
gpio.mode(1,gpio.OUTPUT,gpio.PULLUP)
gpio.serout(1,gpio.HIGH,{30,30,60,60,30,30})  -- serial one byte,
b10110010
gpio.serout(1,gpio.HIGH,{30,70},8)  -- serial 30% pwm 10k, lasts 8 cycles
gpio.serout(1,gpio.HIGH,{3,7},8)  -- serial 30% pwm 100k, lasts 8 cycles
gpio.serout(1,gpio.HIGH,{0,0},8)  -- serial 50% pwm as fast as possible,
lasts 8 cycles
gpio.serout(1,gpio.LOW,{20,10,10,20,10,10,10,100}) -- sim uart one byte
0x5A at about 100kbps
gpio.serout(1,gpio.HIGH,{8,18},8) -- serial 30% pwm 38k, lasts 8 cycles

gpio.serout(1,gpio.HIGH,{5000,995000},100, function() print("done") end) -
- asynchronous 100 flashes 5 ms long every second with a callback function
when done
gpio.serout(1,gpio.HIGH,{5000,995000},100, 1) -- asynchronous 100 flashes
5 ms long, no callback
```

gpio.trig()

Встановіть або очистіть функцію зворотного виклику для запуску при перериванні для PIN-коду.

Ця функція недоступна, якщо GPIO_INTERRUPT_ENABLE не було визначено під час компіляції.

Синтаксис

`gpio.trig(pin, [type [, callback_function]])`

Параметри

- `pin` **1-12** , контакт для запуску, індекс IO. `type` "вгору", "вниз", "обидва", "низький", "високий", які представляють *наростаючий фронт* , *спадаючий фронт* , *обидва фронти* , *низький* і *високий рівні* режимів запуску відповідно. Якщо тип «немає» або опущений, функція зворотного виклику видаляється, а переривання вимикається.
- `callback_function(level, when, eventcount)` функція зворотного виклику, коли виникає тригер.

  Рівень зазначеного піна під час переривання передається як перший параметр зворотного виклику. Мітка часу події передається як другий параметр. Це в мікросекундах і має ту саму основу, що й для `tmr.now()` .

  Ця позначка часу береться на рівні переривання та є більш узгодженою, ніж отримання часу у функції зворотного виклику.

  Ця позначка часу зазвичай є першим виявленим перериванням, але в умовах перевантаження може бути пізнішою.

  Кількість подій - це кількість переривань, які були усунені для цього зворотного виклику.

  Це найкраще працює для переривань, викликаних фронтом, і дозволяє підраховувати фронти.

Повернення

`nil`

приклад

```
do
  -- use pin 1 as the input pulse width counter
  local pin, pulse1, du, now, trig = 1, 0, 0, tmr.now, gpio.trig
  gpio.mode(pin,gpio.INT)
  local function pin1cb(level, pulse2)
    print( level, pulse2 - pulse1 )
    pulse1 = pulse2
```

```
    trig(pin, level == gpio.HIGH  and "down" or "up")
  end
  trig(pin, "down", pin1cb)
end
```

Дивись також

`gpio.mode()`

gpio.write()

Встановлює значення цифрового контакту GPIO.

Синтаксис

`gpio.write(pin, level)`

Параметри

- `pin` пін для запису, індекс IO
- `level` `gpio.HIGH` або `gpio.LOW`

Повернення

`nil`

приклад

```
-- set pin index 1 to GPIO mode, and set the pin to high.
pin=1
gpio.mode(pin, gpio.OUTPUT)

gpio.write(pin, gpio.HIGH )
```

Дивись також

- `gpio.mode()`
- `gpio.read()`

gpio.pulse

Це охоплює набір API, які дозволяють генерувати серії імпульсів із точним синхронізацією на кількох контактах. Він схожий на `serout` API, але може обробляти кілька пінів і має кращий контроль часу.

Основна ідея полягає в тому, щоб створити `gpio.pulse` об’єкт, а потім контролювати його за допомогою методів цього об’єкта. `gpio.pulse` Одночасно може бути активним лише один об’єкт.

Об’єкт побудовано з масиву таблиць, де кожна внутрішня таблиця представляє дію, яку потрібно виконати, і час затримки перед переходом до наступної дії.

Одним із способів використання цього є генерація біполярного імпульсу для керування механізмами годинника, де ви хочете (скажімо) імпульс на контакті 1 на парній секунді та імпульс на контакті 2 на непарній секунді. `:getstate` і `:adjust` може використовуватися для підтримки синхронізації імпульсу з годинником RTC (який сам синхронізується з NTP).

Це (коли створено та запущено) просто виконує крок 1 (встановлюючи вихідні контакти, як зазначено), а потім після 100 000 мкс змінюється на крок 2і.

Це змінює вихідні контакти, а потім очікує 100 000 мкс перед тим, як повернутися до кроку 1.

Це має ефект виведення на контакти 1 і контакти 2 прямокутної хвилі 5 Гц, коли контакти знаходяться поза фазою.

Частота буде трохи нижчою за 5 Гц, оскільки це генерується програмним забезпеченням, і маскування переривання може затримати перехід до наступного кроку.

Щоб наблизитися до 5 Гц, потрібно дозволити тривалості кожного кроку трохи змінюватися. Потім це відрегулює довжину кожного кроку, щоб загалом вихід був на рівні 5 Гц.

Крок	Pin 1	Pin 2	Тривалість (мкс)	Діапазон	Наступний крок
1	Високий	Низький	100 000	90 000 - 110 000	2
2	Низький	Високий	100 000	90 000 - 110 000	1

Перетворюючи це в структуру таблиці, як описано нижче, не потрібно вказувати нічого особливого, якщо номер наступного кроку на один більше, ніж поточний крок.

Указуючи крок поза порядком, ви повинні вказати, як часто ви бажаєте, щоб це було виконано.

Кількість ітерацій може досягати приблизно 4 000 000 000 (фактично будь-яке значення, яке вписується в 32-розрядне ціле число без знаку).

Якщо цього недостатньо повторів, то цикли можна вкладати, як показано нижче:

```
{
  { [1] = gpio.HIGH, [2] = gpio.LOW, delay=500 },
  { [1] = gpio.LOW, [2] = gpio.HIGH, delay=500, loop=1, count=1000000000,
min=400, max=600 },
  { loop=1, count=1000000000 }
}
```

Цикл/підрахунок на кроці 2 призведе до виведення 1 000 000 000 імпульсів (на 1 кГц). Це приблизно 11 днів. На цьому етапі він продовжиться до кроку 3, який запускає 11 днів 1 кГц.

Цей процес повторюватиметься 1 000 000 000 разів (а це приблизно 30 мільйонів років).

Модель циклу полягає в тому, що з кожним циклом пов’язана прихована змінна, яка починається зі `count` значення та зменшується на кожній ітерації, поки не досягне нуля, коли потім переходить до наступного кроку.

Якщо керування знову досягає цього циклу, тоді прихована змінна знову скидається до значення `count`.

gpio.pulse.build

Це створює `gpio.pulse` об’єкт із наданого аргументу (таблиця, як описано нижче).

Синтаксис

`gpio.pulse.build(table)`

Параметр

`table` це перегляд як масив інструкцій. Кожна інструкція представлена такою таблицею:

- Усі цифрові клавіші вважаються PIN-кодами. Значення кожного є значенням, яке потрібно встановити у відповідному рядку GPIO. Наприклад, для `{ [1] = gpio.HIGH }` контакту 1 буде встановлено високий рівень.
  Зауважте, що це номер контакту NodeMCU, а *не* номер GPIO ESP8266. Одночасно можна встановити декілька пінів.
  Зауважте, що можна використовувати будь-який дійсний контакт GPIO, включаючи контакт 0.
- `delay` вказує кількість мікросекунд після встановлення значень PIN-коду для очікування до переходу до наступного стану.
  Фактична затримка може бути довшою за це значення залежно від того, чи ввімкнено переривання в кінцевий час. Максимальне значення становить 64 000 000 -- тобто трохи більше хвилини.
- `min` і `max` може використовуватися для вказівки (разом з `delay`), що цей час можна змінювати. Якщо один часовий інтервал перевищено, то додатковий час буде враховано з періоду часу, який визначено `min` або `max`. Фактичний час також можна налаштувати за допомогою `:adjust` API нижче.
- `count` і `loop` дозволяють простий цикл. Коли стан з `count` і `loop` завершується, наступний стан знаходиться в `loop` (за умови, що він `count` не зменшився до нуля). Підрахунок реалізовано як 32-розрядне ціле число без знаку, тобто має діапазон приблизно до 4 000 000 000. Перший стан - це стан 1. Це `loop` схоже на інструкцію goto, оскільки вказує наступну інструкцію, яку потрібно виконати.

Повернення

`gpio.pulse` об'єкт.

приклад

```
gpio.mode(1, gpio.OUTPUT)
gpio.mode(2, gpio.OUTPUT)

pulser = gpio.pulse.build( {
  { [1] = gpio.HIGH, [2] = gpio.LOW, delay=250000 },
  { [1] = gpio.LOW, [2] = gpio.HIGH, delay=250000, loop=1, count=20,
min=240000, max=260000 }
})

pulser:start(function() print ('done') end)
```

Це створить прямокутну хвилю на контактах 1 і 2, але вони будуть точно не по фазі. Через 10 секунд послідовність буде завершена, а контакт 2 буде високим.

Зауважте, що ви *повинні* встановити контакти в режим виведення (gpio.OUTPUT або gpio.OPENDRAIN) перед початком послідовності виведення, інакше нічого не відбудеться.

gpio.pulse:старт

Це запускає операції виведення.

Синтаксис

`pulser:start([adjust, ] callback)`

Параметр

- `adjust` Це кількість мікросекунд, яку потрібно додати до наступного регульованого періоду. Якщо це значення настільки велике, що воно перевищило б затримку `min` або `max`, тоді залишок утримується до наступного періоду налаштування.

- callback Цей зворотній виклик виконується, коли імпульси завершені. Зворотний виклик викликається з тими ж чотирма параметрами, які описані як значення, що повертаються gpio.pulse:getstate.

Повернення

nil

приклад

```
pulser:start(function(pos, steps, offset, now)
             print (pos, steps, offset, now)
             end)
```

gpio.pulse:getstate

Це повертає поточний стан. Ці чотири значення також передаються у функції зворотного виклику.

Синтаксис

pulser:getstate()

Повернення

- position індекс поточного активного стану. Перший стан - це стан 1. Це nil якщо операція виведення завершена.
- steps кількість станів, які були виконані (включно з поточним). Це дозволяє контролювати прогрес, коли є петлі.
- offset це час (у мікросекундах) до наступного переходу стану. Після завершення операції виведення це значення буде негативним.
- now – значення функції tmr.now() в момент offset обчислення.

приклад

```
pos, steps, offset, now = pulser:getstate()
print (pos, steps, offset, now)
```

gpio.pulse:stop

Це призупиняє операцію виведення в майбутньому.

Синтаксис

`pulser:stop([position ,] callback)`

Параметри

- `position` це індекс, на якому варто зупинитися. Зупинка відбувається при вході в цей стан. Якщо не вказано, зупиняється під час наступного переходу стану.
- `callback` викликається (з тими самими аргументами, які повертає `:getstate`), коли операцію було зупинено.

Повернення

`true` якщо зупинка відбудеться.

приклад

```
pulser:stop(function(pos, steps, offset, now)
            print (pos, steps, offset, now)
            end)
```

gpio.pulse:скасувати

Це негайно зупиняє операцію виведення.

Синтаксис

`pulser:cancel()`

Повернення

- `position` індекс поточного активного стану. Перший стан - це стан 1. Це `nil` якщо операція виведення завершена.
- `steps` кількість станів, які були виконані (включно з поточним). Це дозволяє контролювати прогрес, коли є петлі.
- `offset` це час (у мікросекундах) до наступного переходу стану. Після завершення операції виведення це значення буде негативним.

- now – значення функції tmr.now() в момент offset обчислення.

приклад

```
pulser:cancel(function(pos, steps, offset, now)
              print (pos, steps, offset, now)
              end)
```

gpio.pulse:adjust

Це додає (або віднімає) час, який використовуватиметься у випадку min / max затримка. Це корисно, якщо ви намагаєтеся синхронізувати певний стан із певним часом або зовнішньою подією.

Синтаксис

pulser:adjust(offset)

Параметри

- offset це кількість мікросекунд, які будуть використані в наступних min / max затримках. Це перезаписує будь-яке незавершене зміщення.

Повернення

- position індекс поточного активного стану. Перший стан - це стан 1. Це nil якщо операція виведення завершена.
- steps кількість станів, які були виконані (включно з поточним). Це дозволяє контролювати прогрес, коли є петлі.
- offset це час (у мікросекундах) до наступного переходу стану. Після завершення операції виведення це значення буде негативним.
- now – значення функції tmr.now() в момент offset обчислення.

приклад

```
pulser:adjust(177)
```

gpio.pulse:оновлення

Це може змінити вміст певного кроку в програмі виводу. Це можна використовувати для налаштування часу затримки або навіть для зміни контактів.

Це не можна використовувати для видалення записів або додавання нових записів. Зміна `count` змінює початкове значення, але не змінює поточне значення зменшення;

Синтаксис

`pulser:update(entrynum, entrytable)`

Параметри

- `entrynum` це номер запису у вихідному визначенні послідовності імпульсів. Перший запис має номер 1.
- `entrytable` є таблицею, яка містить ті самі ключі, що й для `gpio.pulse.build`

Повернення

нічого

приклад

```
pulser:update(1, { delay=1000 })
```

МОДУЛЬ BIT

Підтримка бітових маніпуляцій для 32-бітних цілих чисел.

<table>
<tr><td>bit.arshift()</td><td>Арифметичний зсув вправо число, еквівалентне значенню >> зсув у C.</td></tr>
<tr><td>bit.band()</td><td>Побітове І, еквівалентне val1 & val2 & .</td></tr>
<tr><td>bit.bit()</td><td>Згенерувати число з 1 бітом (використовується для створення маски).</td></tr>
<tr><td>bit.bnot()</td><td>Порозрядне заперечення, еквівалент ~значення в C.</td></tr>
<tr><td>bit.bor()</td><td>Порозрядне АБО, еквівалентне val1 | val2 | .</td></tr>
<tr><td>bit.bxor()</td><td>Побітове XOR, еквівалентне val1 ^ val2 ^ .</td></tr>
<tr><td>bit.clear()</td><td>Очистити біти в числі.</td></tr>
<tr><td>bit.isclear()</td><td>Перевірте, чи заданий біт очищено.</td></tr>
<tr><td>bit.isset()</td><td>Перевірте, чи заданий біт встановлено.</td></tr>
<tr><td>bit.lshift()</td><td>Зсув вліво число, еквівалентне зсуву значення << в C.</td></tr>
<tr><td>bit.rshift()</td><td>Логічний зсув вправо числа, еквівалентного значенню (без знаку) >> зсуву в C.</td></tr>
<tr><td>bit.set()</td><td>Установити біти в число.</td></tr>
</table>

ОПИС ФУНКЦІЙ API МОДУЛЯ BIT

bit.arshift()

Арифметичний зсув вправо числа, еквівалентного `value >> shift` C.

Синтаксис

`bit.arshift(value, shift)`

Параметри

- `value` значення для зсуву

- `shift` позиції для зміни

Повернення

число, зміщене вправо (арифметично)

приклад

```
bit.arshift(3, 1) -- returns 1
-- Using a 4 bits representation: 0011 >> 1 == 0001
```

bit.band()

Порозрядне I, еквівалент `val1 & val2 & ... & valn` C.

Синтаксис

`bit.band(val1, val2 [, ... valn])`

Параметри

- `val1` перший аргумент I
- `val2` другий аргумент I
- `...valn` ...n-й аргумент I

Повернення

порозрядне I всіх аргументів (число)

приклад

```
bit.band(3, 2) -- returns 2
-- Using a 4 bits representation: 0011 & 0010 == 0010
```

bit.bit()

Згенерувати число з 1 бітом (використовується для створення маски).

Еквівалент `1 << position` C.

Синтаксис

`bit.bit(position)`

Параметри

`position` положення біта, який буде встановлено на 1

Повернення

число лише з одним бітом 1 на позиції (решта встановлюється на 0)

приклад

```
bit.bit(4) -- returns 16
```

bit.bnot()

Порозрядне заперечення, еквівалент `~value in C.`

Синтаксис

`bit.bnot(value)`

Параметри

`value` число для заперечення

Повернення

порозрядне значення числа

bit.bor()

Порозрядне АБО, еквівалентно `val1 | val2 | ... | valn` C.

Синтаксис

`bit.bor(val1, val2 [, ... valn])`

Параметри

- `val1` перший аргумент АБО.
- `val2` другий аргумент АБО.
- `...valn` ...n-й аргумент АБО

Повернення

порозрядне АБО всіх аргументів (число)

приклад

```
bit.bor(3, 2) -- returns 3
-- Using a 4 bits representation: 0011 | 0010 == 0011
```

bit.bxor()

Побітове XOR, еквівалентно `val1 ^ val2 ^ ... ^ valn` C.

Синтаксис

`bit.bxor(val1, val2 [, ... valn])`

Параметри

- `val1` перший аргумент XOR
- `val2` другий аргумент XOR
- `...valn` ...n-й аргумент XOR

Повернення

побітове XOR усіх аргументів (число)

приклад

```
bit.bxor(3, 2) -- returns 1
-- Using a 4 bits representation: 0011 ^ 0010 == 0001
```

bit.clear()

Очистити біти в числі.

Синтаксис

`bit.clear(value, pos1 [, ... posn])`

Параметри

- `value` базове число
- `pos1` позиція першого біта для очищення
- `...posn` позиція n-го біта для очищення

Повернення

число з очищеним бітом(ами) у вказаній позиції(ях)

приклад

```
bit.clear(3, 0) -- returns 2
```

bit.isclear()

Перевірте, чи заданий біт очищено.

Синтаксис

`bit.isclear(value, position)`

Параметри

- `value` значення для перевірки
- `position` положення біта для перевірки

Повернення

істина, якщо біт у заданій позиції дорівнює 0, хибність в іншому випадку

приклад

```
bit.isclear(2, 0) -- returns true
```

bit.isset()

Перевірте, чи заданий біт встановлено.

Синтаксис

`bit.isset(value, position)`

Параметри

- `value` значення для перевірки
- `position` положення біта для перевірки

Повернення

істина, якщо біт у заданій позиції дорівнює 1, хибність у протилежному випадку

приклад

```
bit.isset(2, 0) -- returns false
```

bit.lshift()

Зміщення вліво число, еквівалентне `value << shift` C.

Синтаксис

`bit.lshift(value, shift)`

Параметри

- `value` значення для зсуву
- `shift` позиції для зміни

Повернення

число змістилося вліво

приклад

```
bit.lshift(2, 2) -- returns 8
-- Using a 4 bits representation: 0010 << 2 == 1000
```

bit.rshift()

Логічний зсув вправо числа, еквівалентного `( unsigned )value >> shift` C.

Синтаксис

`bit.rshift(value, shift)`

Параметри

- `value` значення для зсуву.
- `shift` позиції для зміни.

Повернення

число зміщене праворуч (логічно)

приклад

```
bit.rshift(2, 1) -- returns 1
-- Using a 4 bits representation: 0010 >> 1 == 0001
```

bit.set()

Установити біти в число.

Синтаксис

`bit.set(value, pos1 [, ... posn ])`

Параметри

- `value` базове число.
- `pos1` положення першого біта для встановлення.
- `...posn` положення n-го біта для встановлення.

Повернення

число з бітами, встановленими в даній позиції

приклад

```
bit.set(2, 0) -- returns 3
```

ФАЙЛОВА СИСТЕМА SPIFFS

ESP8266 використовує файлову систему SPIFFS для зберігання файлів у флеш-чіпі. Технічні відомості про те, як це налаштовано, можна знайти нижче, а також різні параметри часу збірки.

spiffsimg - маніпулювання образами дисків SPI Flash File System

Синтаксис

```
spiffsimg -f <filename>
    [-o <offsetfile>]
    [-c <size>]
    [-S <flashsize>]
    [-U <usedsize>]
    [-d]
    [-l | -i | -r <scriptname> ]
```

Підтримувані операції:

- -f визначає ім'я файлу для образу диска. '%x' буде замінено обчисленим зміщенням файлової системи (-U також необхідно вказати, щоб обчислити зміщення).
- -o вказує назву файлу, який має містити обчислене зміщення.
- -S визначає розмір мікросхеми флеш-пам'яті. 32m становить 32 біт, 4MB становить 4 мегабайти.
- -U визначає обсяг флеш-пам'яті, який використовується мікропрограмою. Десяткові чи шістнадцяткові байти (якщо починається з 0x).
- -c Створіть чистий образ диска заданого розміру. Десяткові чи шістнадцяткові байти (якщо починається з 0x).
- -l Перелічіть вміст даного образу диска.
- -i Інтерактивні команди.

- -r Скриптові команди з імені файлу.
- -d призводить до видалення образу диска в разі помилки. Це полегшує створення сценарію.

Доступні команди:

- ls Список вмісту. Формат виведення: {type} {size} {name}.
- cat <filename> Дамп вмісту файлу до stdout.
- rm <filename> Видалити файл.
- info Показати оцінки використання SPIFFS.
- import <srcfile> <spiffsname> Імпортуйте файл в образ диска.
- export <spiffsname> <dstfile> Експорт файлу з образу диска.

приклад:

```
# spiffsimg -f flash.img -S 32m -U 524288 -i
> import myapp/lua/init.lua init.lua
> import myapp/lua/httpd.lua httpd.lua
> import myapp/html/index.html http/index.html
> import myapp/html/favicon.ico http/favicon.ico
> ls
f    122 init.lua
f   5169 httpd.lua
f   2121 http/index.html
f    880 http/favicon.ico
>^D
#
```

Відомі обмеження:

- Розміри блоків і сторінок жорстко закодовані, щоб бути сумісними з nodemcu.
- Обробка помилок не зовсім узгоджена, деякі помилки призводять до раннього виходу, інші просто друкують помилку (однак обидва викликають ненульовий вихід).
- Підтримуються лише плоскі SPIFFS.

Технічні деталі

Конфігурація SPIFFS складається з 4k секторів (єдиний розмір, який підтримує SDK) і 8k блоків. 256 байт сторінок. Magic увімкнено, і magic_len також увімкнено.

Це дозволяє мікропрограмі знайти початок файлової системи (а також розмір). Однією з цілей є зробити файлову систему більш стійкою під час оновлення прошивки.

Проте все ще є випадки, коли spiffs виявляє файлову систему та використовує її, якщо вона недійсна. Існує два значних розміри флеш-пам'яті - 512K і 4M (або більше).

Файлова система має починатися на межі 4 Кб, але оскільки вона закінчується на набагато більшій межі (межі 16 Кб), вона також починається на межі 8 Кб.

Для невеликої мікросхеми флеш-пам'яті небагато вільного місця, тому щойно відформатована файлова система запускатиметься якомога менше (щоб отримати якомога більше місця).

Для великої флеш-пам'яті файлова система розпочнеться з межі 64 КБ. Щойно відформатована файлова система починатиметься від 64 Кб до 128 Кб після завершення мікропрограми.

Це означає, що файлова система витримає багато перепрошивок і принаймні 64 Кб зростання прошивки.

Стандартний процес збирання мікропрограми збирає `spiffsimg` інструмент (знаходиться у `tools/spiffsimg` підкаталозі).

Makefile верхнього рівня також перевіряє наявність даних у `local/fs` дереві каталогів, а потім копіює ці файли в образ флеш-диска.

Зазвичай буде створено два зображення - одне для частини флеш-пам'яті 512 КБ, інше - для частини флеш-пам'яті 4 МБ.

Якщо дані не вписуються в частину 512k після включення мікропрограми, то файл не буде згенеровано.

Файл образу диска розміщується в bin каталозі та має назву, 0x<offset>-<size>.bin де зсув - це місце, де його слід прошивати, а розмір - розмір частини флеш-пам’яті.

Цілком допустимо (і швидше) прошити зображення 512k у частину 4M. Проте, ймовірно, у файловій системі буде обмежений простір для створення нових файлів.

Конфігурація за замовчуванням намагатиметься створити три різні файлові системи для розмірів флеш-пам’яті 512 КБ, 1 МБ та 4 МБ.

Розмір 1 МБ підходить для ESP8285. Це можна змінити, вказавши параметр FLASHSIZE у make-файлі.

Якщо local/fs каталог порожній, флеш-зображення не створюватимуться (а зображення з останньої збірки буде видалено). Потім інструмент spiffsimg можна використовувати для створення потрібного зображення.

Якщо під час завантаження платформи не знайдено жодної файлової системи, буде відформатовано нову файлову систему. Це може зайняти деякий час під час першого завантаження.

Щоб пришвидшити час завантаження, можна визначити (під час збирання) розмір файлової системи SPIFFS, яку потрібно відформатувати.

Це може бути лише 32768 байт, що дає файловій системі приблизно 15 Кбайт корисного простору.

```
#define SPIFFS_MAX_FILESYSTEM_SIZE  32768
```

Це обмеження розміру файлової системи впливає лише на форматування файлової системи – якщо фірма знайде існуючу дійсну файлову систему (будь-якого розміру), вона використовуватиме її. Однак, якщо файлову систему переформатувати з Lua (за допомогою file.format()), нова файлова система відповідатиме обмеженню розміру.

Також є можливість керувати розташуванням файлової системи SPIFFS:

```
#define SPIFFS_FIXED_LOCATION       0x100000
```

Це вказує на те, що файлова система SPIFFS починається з 1 Мб від початку флеш-пам’яті. Якщо не вказано інше, він працюватиме до кінця флеш-пам’яті (за винятком 16 КБ місця, зарезервованого SDK).

Існує параметр, який обмежує розмір файлової системи до наступної межі 1 МБ (мінус 16 Кб для простору параметрів). Це може бути корисним під час оновлення OTA.

```
#define SPIFFS_SIZE_1M_BOUNDARY
```

ФАЙЛОВИЙ МОДУЛЬ

Файловий модуль забезпечує доступ до файлової системи та її окремих файлів.

Файлова система - це плоска файлова система без поняття підкаталогів/папок.

Окрім файлової системи SPIFFS на внутрішній флеш-пам’яті, цей модуль також може отримати доступ до розділів FAT на зовнішній SD-карті, якщо FatFS увімкнено .

```
-- open file in flash:
if file.open("init.lua") then
  print(file.read())
  file.close()
end

-- or with full pathspec
file.open("/FLASH/init.lua")

-- open file on SD card
if file.open("/SD0/somefile.txt") then
  print(file.read())
  file.close()
end
```

file.chdir()	Змінити поточний каталог (і диск).
file.exists()	Визначає, чи існує вказаний файл.
file.format()	Відформатувати файлову систему.
file.fscfg ()	Повертає флеш-адресу та фізичний розмір області файлової системи в байтах.
file.fsinfo()	Повертає інформацію про розмір файлової системи.
file.getcontents()	Відкрийте та прочитайте вміст файлу.
file.list()	Перераховує всі файли у файловій системі.
file.mount()	Встановлює том FatFs на SD-карту.
file.on()	Реєструє функції зворотного виклику.

file.open()	Відкриває файл для доступу, потенційно створює його (для режимів запису).
file.remove()	Видалити файл із файлової системи.
file.putcontents()	Відкрити та записати вміст файлу.
file.rename()	Перейменовує файл.
file.stat()	Отримати атрибути файлу або каталогу в таблиці.
Базова модель	У базовій моделі одночасно відкривається максимум один файл.
Об'єктна модель	Файли представлені файловими об’єктами, створеними файлом.
file.close(), file.obj:close()	Закриває відкритий файл, якщо такий є.
file.flush(), file.obj:flush()	Очищає всі незавершені записи у файлову систему, гарантуючи, що жодні дані не будуть втрачені під час перезапуску.
file.read(), file.obj:read()	Читання вмісту відкритого файлу.
file.readline(), file.obj:readline()	Читання наступний рядок із відкритого файлу.
file.seek(), file.obj:seek()	Встановлює та отримує позицію файлу, виміряну від початку файлу, до позиції, визначеної зміщенням плюс базу, визначену рядком where.
file.write(), file.obj:write()	Записати рядок у відкритий файл.
file.writeline(), file.obj:writeline()	Записати рядок у відкритий файл і додайте '\n' у кінці.

ОПИС ФУНКЦІЙ API МОДУЛЯ FILE

file.chdir()

Змінити поточний каталог (і диск). Це буде використано, якщо диск/каталог не додається до імен файлів.

Поточний каталог за замовчуванням є кореневим внутрішнім SPIFFS (`/FLASH`) після запуску системи.

Примітка

Функція доступна лише тоді, коли підтримка FatFS скомпільована в мікропрограму.

Синтаксис

`file.chdir(dir)`

Параметри

`dir` назва каталогу - `/FLASH`, `/SD0`, `/SD1` і т.д.

Повернення

`true` на успіх, `false` інакше

file.exists()

Визначає, чи існує вказаний файл.

Синтаксис

`file.exists(filename)`

Параметри

- `filename` файл для перевірки

Повернення

true, якщо файл існує (навіть якщо розмір 0 байт), і false, якщо він не існує

приклад

```
files = file.list()
if files["device.config"] then
    print("Config file exists")
end

if file.exists("device.config") then
    print("Config file exists")
end
```

Дивись також

file.list()

file.format()

Форматує файлову систему. Повністю стирає будь-яку існуючу файлову систему та записує нову. Залежно від розміру мікросхеми флеш-пам’яті в ESP, це може зайняти кілька секунд.

Примітка

Функція не підтримується для SD-карт.

Синтаксис

file.format()

Параметри

немає

Повернення

nil

Дивись також

file.remove()

file.fscfg ()

Повертає флеш-адресу та фізичний розмір області файлової системи в байтах.

Примітка

Функція не підтримується для SD-карт.

Синтаксис

`file.fscfg()`

Параметри

немає

Повернення

- `flash address` (число)
- `size` (число)

приклад

```
print(string.format("0x%x", file.fscfg()))
```

file.fsinfo()

Повертає інформацію про розмір файлової системи. Одиницею вимірювання є байт для SPIFFS і кбайт для FatFS.

Синтаксис

`file.fsinfo()`

Параметри

немає

Повернення

- `remaining` (число)
- `used` (число)
- `total` (число)

приклад

```
-- get file system info
remaining, used, total=file.fsinfo()
```

```
print("\nFile system info:\nTotal : "..total.." (k)Bytes\nUsed :
"..used.." (k)Bytes\nRemain: "..remaining.." (k)Bytes\n")
```

file.getcontents()

Відкрийте та прочитайте вміст файлу.

Синтаксис

`file.getcontents(filename)`

Параметри

- `filename` файл, який потрібно відкрити та прочитати

Повернення

вміст файлу, якщо файл існує. `nil` якщо файл не існує.

Приклад (базова модель)

```
print(file.getcontents('welcome.txt'))
```

Дивись також

- `file.putcontents()`

file.list()

Перераховує всі файли у файловій системі.

Синтаксис

`file.list([pattern])`

Параметри

- `pattern` буде повернуто лише файли, що відповідають шаблону Lua

Повернення

таблиця Lua, яка містить усі пари {ім'я файлу: розмір файлу}, якщо не вказано шаблон. Якщо вказано шаблон, лише ті імена файлів, які відповідають шаблону (інтерпретуються як традиційний шаблон Lua , а не, скажімо, глобус оболонки UNIX), будуть включені до результуючої таблиці. `file.list` викличе будь-які помилки, виявлені під час зіставлення шаблону.

приклад

```
l = file.list();
for k,v in pairs(l) do
  print("name:"..k..", size:"..v)
end
```

file.mount()

Встановлює том FatFs на SD-карту.

Примітка

Функція доступна лише тоді, коли підтримка FatFS скомпільована в мікропрограму та не підтримується для внутрішньої флеш-пам'яті.

Синтаксис

`file.mount(ldrv[, pin])`

Параметри

- `ldrv` ім'я логічного диска, `/SD0`, `/SD1` тощо.
- `pin` 1~12, індекс IO для SS/CS, за замовчуванням 8, якщо опущено.

Повернення

Об'ємний об'єкт

приклад

```
vol = file.mount("/SD0")
vol:umount()
```

file.on()

Реєструє функції зворотного виклику.

Тригерними подіями є:

- `rtc` доставляти поточну дату й час у файлову систему. Очікується, що функція повертатиме таблицю, що містить поля `year`, `mon`, `day`, `hour`, `min`, `sec` поточної дати та часу. Не підтримується для внутрішнього спалаху.

Синтаксис

`file.on(event[, function()])`

Параметри

- `event` рядок
- `function()` функція зворотного виклику. Скасовує реєстрацію зворотного виклику, якщо `function()` пропущено або `nil`.

Повернення

`nil`

приклад

```
sntp.sync(server_ip,
  function()
    print("sntp time sync ok")
    file.on("rtc",
      function()
        return rtctime.epoch2cal(rtctime.get())
      end)
  end)
```

Дивись також

`rtctime.epoch2cal()`

file.open()

Відкриває файл для доступу, потенційно створює його (для режимів запису).

Після завершення роботи з файлом його потрібно закрити за допомогою `file.close()`.

Синтаксис

`file.open(filename, mode)`

Параметри

- `filename` файл, який потрібно відкрити
- `mode`:
  - "r": режим читання (за замовчуванням)

- "w": режим запису
- "a": режим додавання
- "r+": режим оновлення, усі попередні дані зберігаються
- "w+": режим оновлення, усі попередні дані видаляються
- "a+": режим додавання оновлення, попередні дані зберігаються, запис дозволяється лише в кінці файлу

Повернення

об'єкт файлу, якщо файл відкрито нормально. `nil` якщо файл не відкрито або не існує (режими читання).

Приклад (базова модель)

```
-- open 'init.lua', print the first line.
if file.open("init.lua", "r") then
  print(file.readline())
  file.close()
end
```

Приклад (об'єктна модель)

```
-- open 'init.lua', print the first line.
fd = file.open("init.lua", "r")
if fd then
  print(fd:readline())
  fd:close(); fd = nil
end
```

Дивись також

- `file.close()`
- `file.readline()`

file.remove()

Видалити файл із файлової системи. Файл не має бути відкритим.

Синтаксис

`file.remove(filename)`

Параметри

`filename` файл для видалення

Повернення

`nil`

приклад

```
-- remove "foo.lua" from file system.
file.remove("foo.lua")
```

Дивись також

`file.open()`

file.putcontents()

Відкрити та записати вміст файлу.

Синтаксис

`file.putcontents(filename, contents)`

Параметри

- `filename` файл, який буде створено
- `contents` для запису у файл

Повернення

`true` якщо запис в порядку, `nil` на помилку

Приклад (базова модель)

```
file.putcontents('welcome.txt', [[
  Hello to new user
  -----------------
]])
```

Дивись також

- `file.getcontents()`

file.rename()

Перейменовує файл. Якщо файл зараз відкритий, він буде закритий першим.

Синтаксис

`file.rename(oldname, newname)`

Параметри

- `oldname` старе ім'я файлу
- `newname` нове ім'я файлу

Повернення

`true` на успіх, `false` на помилку.

приклад

```
-- rename file 'temp.lua' to 'init.lua'.
file.rename("temp.lua","init.lua")
```

file.stat()

Отримати атрибути файлу або каталогу в таблиці. Елементами таблиці є:

- `size` розмір файлу в байтах
- `name` ім'я файлу
- `time` таблиця з інформацією про позначку часу. За замовчуванням 1970-01-01 00:00:00, якщо мітки часу не підтримуються (на SPIFFS).
  - `year`
  - `mon`
  - `day`
  - `hour`
  - `min`
  - `sec`

- is_dir прапорець, true якщо елемент є каталогом, інакше false
- is_rdonly прапорець, true якщо елемент доступний лише для читання, інакше false
- is_hidden прапорець true, якщо елемент приховано, інакше false
- is_sys прапорець, true якщо елемент системний, інакше false
- is_arch прапорець true, якщо елемент є архівним, інакше false

Синтаксис

file.stat(filename)

Параметри

filename ім'я файлу

Повернення

таблиця, що містить атрибути файлів

приклад

```
s = file.stat("/SD0/myfile")
print("name: " .. s.name)
print("size: " .. s.size)

t = s.time
print(string.format("%02d:%02d:%02d", t.hour, t.min, t.sec))
print(string.format("%04d-%02d-%02d", t.year, t.mon, t.day))

if s.is_dir then print("is directory") else print("is file") end
if s.is_rdonly then print("is read-only") else print("is writable") end
if s.is_hidden then print("is hidden") else print("is not hidden") end
if s.is_sys then print("is system") else print("is not system") end
if s.is_arch then print("is archive") else print("is not archive") end

s = nil
t = nil
```

ФУНКЦІЇ ДОСТУПУ ДО ФАЙЛІВ

Модуль `file` надає кілька функцій для доступу до вмісту файлу після його відкриття за допомогою `file.open()`. Їх можна використовувати як частину базової моделі або об'єктної моделі:

Базова модель

У базовій моделі одночасно відкривається максимум один файл. Функції доступу до файлу працюють із цим файлом за замовчуванням. Якщо відкривається інший файл, попередньо потрібно закрити попередній файл за замовчуванням.

```
-- open 'init.lua', print the first line.
if file.open("init.lua", "r") then
  print(file.readline())
  file.close()
end
```

Об'єктна модель

Файли представлені файловими об'єктами, створеними за допомогою `file.open()`. Функції доступу до файлів доступні як методи цього об'єкта, і кілька файлових об'єктів можуть існувати одночасно.

```
src = file.open("init.lua", "r")
if src then
  dest = file.open("copy.lua", "w")
  if dest then
    local line
    repeat
      line = src:read()
      if line then
        dest:write(line)
      end
    until line == nil
    dest:close(); dest = nil
  end
  src:close(); dest = nil
end
```

Увага

Рекомендується використовувати лише одну модель у програмі. Одночасне використання обох моделей може призвести до непередбачуваної поведінки: закриття файлу за замовчуванням із базової моделі також закриє відповідний файловий об’єкт. Закриття файлу з об’єктної моделі також закриє файл за замовчуванням, якщо це той самий файл.

Примітка

Максимальна кількість відкритих файлів у SPIFFS визначається під час компіляції за SPIFFS_MAX_OPEN_FILESдопомогою user_config.h.

file.close(), file.obj:close()

Закриває відкритий файл, якщо такий є.

Синтаксис

`file.close()`

`fd:close()`

Параметри

немає

Повернення

`nil`

Дивись також

`file.open()`

file.flush(), file.obj:flush()

Очищає всі незавершені записи у файлову систему, гарантуючи, що жодні дані не будуть втрачені під час перезапуску. Закриття відкритого файлу за допомогою `file.close()` / `fd:close()` також виконує неявне очищення.

Синтаксис

`file.flush()`

fd:flush()

Параметри

немає

Повернення

nil

Приклад (базова модель)

```
-- open 'init.lua' in 'a+' mode
if file.open("init.lua", "a+") then
  -- write 'foo bar' to the end of the file
  file.write('foo bar')
  file.flush()
  -- write 'baz' too
  file.write('baz')
  file.close()
end
```

Дивись також

file.close() / file.obj:close()

file.read(), file.obj:read()

Читання вмісту відкритого файлу.

Примітка

Функція тимчасово виділяє 2 * (кількість запитуваних байтів) у купі для буферизації та обробки прочитаних даних. Розмір блоку за замовчуванням (FILE_READ_CHUNK) становить 1024 байти і вважається безпечним. Збільшення цього значення в 4 рази або більше може призвести до переповнення купи залежно від програми.

Синтаксис

file.read([n_or_char])

fd:read([n_or_char])

Параметри

- n_or_char :

- якщо нічого не передано, тоді прочитати до FILE_READ_CHUNK байтів або весь файл (залежно від того, що менше).
- якщо передано число n, то прочитати до n байтів або весь файл (залежно від того, що менше).
- якщо передано рядок, що містить один символ char, тоді читати, доки не char з’явиться наступний у файлі, FILE_READ_CHUNK не буде прочитано байти або досягнуто EOF.

Повернення

Вміст файлу у вигляді рядка або нуль, коли EOF

Приклад (базова модель)

```
-- print the first line of 'init.lua'
if file.open("init.lua", "r") then
  print(file.read('\n'))
  file.close()
end
```

Приклад (об'єктна модель)

```
-- print the first 5 bytes of 'init.lua'
fd = file.open("init.lua", "r")
if fd then
  print(fd:read(5))
  fd:close(); fd = nil
end
```

Дивись також

- file.open()
- file.readline() / file.obj:readline()

file.readline(), file.obj:readline()

Читати наступний рядок із відкритого файлу. Рядки визначаються як нуль або більше байтів, що закінчуються байтом EOL ('\n'). Якщо наступний рядок довший за 1024, ця функція повертає лише перші 1024 байти.

Синтаксис

file.readline()

fd:readline()

Параметри

немає

Повернення

Вміст файлу в рядку, рядок за рядком, включаючи EOL('\n').

Повернутися nil, коли EOF.

Приклад (базова модель)

```
-- print the first line of 'init.lua'
if file.open("init.lua", "r") then
  print(file.readline())
  file.close()
end
```

Дивись також

- file.open()
- file.close() / file.obj:close()
- file.read() / file.obj:read()

file.seek(), file.obj:seek()

Встановлює та отримує позицію файлу, виміряну від початку файлу, до позиції, визначеної зміщенням плюс базу, визначену рядком where.

Синтаксис

file.seek([whence [, offset]])

fd:seek([whence [, offset]])

Параметри

- whence
  - "set": основа - позиція 0 (початок файлу)
  - "cur": база є поточною позицією (значення за замовчуванням)
  - "кінець": база є кінцем файлу
- offset за замовчуванням 0

Якщо параметри не вказано, функція просто повертає поточне зміщення файлу.

Повернення

результуюча позиція файлу або nil помилка

Приклад (базова модель)

```
if file.open("init.lua", "r") then
  -- skip the first 5 bytes of the file
  file.seek("set", 5)
  print(file.readline())
  file.close()
end
```

Дивись також

file.open()

file.write(), file.obj:write()

Записати рядок у відкритий файл.

Синтаксис

file.write(string)

fd:write(string)

Параметри

string вміст для запису у файл

Повернення

true якщо запис в порядку, nil на помилку

Приклад (базова модель)

```
-- open 'init.lua' in 'a+' mode
if file.open("init.lua", "a+") then
  -- write 'foo bar' to the end of the file
  file.write('foo bar')
  file.close()
end
```

Приклад (об'єктна модель)

```
-- open 'init.lua' in 'a+' mode
fd = file.open("init.lua", "a+")
if fd then
  -- write 'foo bar' to the end of the file
  fd:write('foo bar')
  fd:close()
end
```

Дивись також

- file.open()
- file.writeline() / file.obj:writeline()

file.writeline(), file.obj:writeline()

Напишіть рядок у відкритий файл і додайте '\n' у кінці.

Синтаксис

file.writeline(string)

fd:writeline(string)

Параметри

string вміст для запису у файл

Повернення

true якщо писати добре, nil на помилку

Приклад (базова модель)

```
-- open 'init.lua' in 'a+' mode
if file.open("init.lua", "a+") then
  -- write 'foo bar' to the end of the file
  file.writeline('foo bar')
  file.close()
end
```

Дивись також

file.open()

file.readline()/file.obj:readline()

3. API ПРОГРАМУВАННЯ МЕРЕЖЕВИХ МОДУЛІВ ESP8266

МОДУЛЬ NODE

Модуль вузла надає доступ до функцій системного рівня, таких як сплячий режим, перезапуск і різна інформація та ідентифікатори.

node.bootreason()	Повертає причину завантаження та розширену інформацію про скидання.
node.chipid()	Повертає ідентифікатор чіпа ESP.
node.compile()	Компілює текстовий файл Lua у байт-код Lua та зберігає його як .
node.dsleep()	Переходить у режим глибокого сну, прокидається після закінчення часу.
node.dsleepMax()	Повертає поточну теоретичну максимальну тривалість глибокого сну.
node.flashid()	Повертає ідентифікатор флеш-чіпа.
node.flashindex()	Застарілий синонім вузла.
node.flashreload()	Застарілий синонім вузла.
node.flashsize()	Повертає розмір мікросхеми флеш-пам’яті в байтах.
node.getcpufreq()	Повертає поточну частоту ЦП.
node.getpartitiontable()	Повертає поточну інформацію про розділи LFS і SPIFFS.
node.heap()	Повертає поточний доступний розмір купи в байтах.

node.info()	Повертає інформацію про апаратне забезпечення, версію програмного забезпечення та конфігурацію збірки.
node.input()	Надсилає рядок інтерпретатору Lua.
node.LFS	Підтаблиця, що містить API для доступу до Lua Flash Store (LFS).
node.LFS.get()	Повертає посилання на функцію для функції в LFS.
node.LFS.list()	Перелічіть модулі в LFS.
node.LFS.reload()	Перезавантажте LFS із наданим флеш-образом.
node.output()	Перенаправляє інтерпретатор Lua на канал стандартного виводу, коли вказано функцію CB (див. модуль каналу), а в іншому випадку повертає вихід до нормального.
node.readvdd33() --застаріло	Переміщено в adc.
node.restart()	Перезапускає чіп.
node.restore()	
node.setcpufreq()	Замінює робочу частоту ЦП.
node.setonerror()	Замінює стандартну обробку збоїв, яка завжди перезавантажує систему.

node.setpartitiontable()	Встановлює поточну інформацію про розділ LFS та/або SPIFFS.
node.sleep()	Переведіть NodeMCU в режим легкого сну, щоб зменшити споживання струму.
node.startupcommand()	Замінює дію запуску за замовчуванням під час перезавантаження процесора, замінюючи виконання init.
node.startupcounts()	Запитати продуктивність запуску системи.
node.startup()	Отримує/ Встановлює параметри, які керують процесом запуску.
node.stripdebug()	Контролює обсяг налагоджувальної інформації, що зберігається під час вузла.
node.osprint()	Контролює, чи друкуються результати налагодження з Espressif SDK.
node.random()	Це поводиться як математика.
node.egc.setmode()	Встановлює режим аварійного збирача сміття.
node.egc.meminfo()	Повертає інформацію про використання пам’яті для середовища виконання Lua.
node.task.post()	Увімкніть зворотний виклик Lua або завдання, щоб опублікувати інший запит завдання.

ОПИС ФУНКЦІЙ API МОДУЛЯ NODE

node.bootreason()

Повертає причину завантаження та розширену інформацію про скидання.

Перше значення, яке повертається, - це необроблений код, а не новий код «скидання інформації», який було представлено в останніх SDK. Цінності:

- 1, увімкнення
- 2, скинути (програмне забезпечення?)
- 3, апаратне скидання через контакт скидання
- 4, скидання WDT (тайм-аут сторожового таймера)

Друге повернуте значення є розширеною причиною скидання. Цінності:

- 0, увімкнення
- 1, скидання апаратного сторожового таймера
- 2, скидання винятків
- 3, скидання сторожового програмного забезпечення
- 4, перезапуск програмного забезпечення
- 5, прокинутися від глибокого сну
- 6, зовнішнє скидання

Загалом розширена причина скидання замінює необроблений код. Необроблений код зберігається лише для зворотної сумісності. Для нових програм настійно рекомендується замість цього використовувати розширену причину скидання.

У разі причини розширеного скидання 3 (скидання виняткової ситуації) повертаються додаткові значення, що містять інформацію про збій. Це, по порядку, EXCCAUSE , EPC1, EPC2, EPC3, EXCVADDR і DEPC.

Синтаксис

```
node.bootreason()
```

Параметри

немає

Повернення

```
rawcode, reason [, exccause, epc1, epc2, epc3, excvaddr, depc ]
```

приклад

```
_, reset_reason = node.bootreason()
if reset_reason == 0 then print("Power UP!") end
```

node.chipid()

Повертає ідентифікатор чіпа ESP.

Синтаксис

```
node.chipid()
```

Параметри

немає

Повернення

ID чіпа (номер)

node.compile()

Компілює текстовий файл Lua у байт-код Lua та зберігає його як файл .lc.

Синтаксис

```
node.compile("file.lua")
```

Параметри

`filename` ім'я текстового файлу Lua

Повернення

```
nil
```

приклад

```
file.open("hello.lua","w+")
file.writeline([[print("hello nodemcu")]])
file.writeline([[print(node.heap())]])
file.close()
```

```
node.compile("hello.lua")
dofile("hello.lua")
dofile("hello.lc")
```

node.dsleep()

Переходить у режим глибокого сну, прокидається після закінчення часу.

Обережно

Цю функцію можна використовувати лише за умови, що esp8266 PIN32(RST) і PIN8(XPD_DCDC aka GPIO16) з'єднані разом. Використання sleep(0) не встановить таймер пробудження, під'єднайте GPIO до контакту RST, чіп прокинеться через спадаючий фронт на контакті RST.

Синтаксис

`node.dsleep(us, option, instant)`

Параметри

- `us` число (ціле) або `nil`, час сну в мікросекундах. Якщо `us == 0`, то спатиме вічно. Якщо `us == nil`, час сну не буде встановлено.
- `option` число (ціле) або `nil`. Якщо `nil`, буде використано параметр останнього живого як параметр за замовчуванням.
  - 0, байт початкових даних 108 є цінним
  - > 0, байт початкових даних 108 не має значення
  - 0, RF_CAL чи ні після пробудження глибокого сну, залежить від 108 байту даних ініціалізації
  - 1, RF_CAL після пробудження глибокого сну буде великий струм
  - 2, немає RF_CAL після пробудження глибокого сну, буде лише невеликий струм
  - 4, вимкніть радіочастоту після пробудження глибокого сну, як і модем, буде найменший струм
- `instant` число (ціле) або `nil`. Якщо він присутній і відмінний від нуля, мікросхема негайно перейде в режим глибокого сну і не чекатиме, поки ядро Wi-Fi вимкнеться.

Повернення

nil

приклад

```
--do nothing
node.dsleep()
--sleep µs
node.dsleep(1000000)
--set sleep option, then sleep µs
node.dsleep(1000000, 4)
--set sleep option only
node.dsleep(nil,4)
```

Дивись також

- wifi.suspend()
- wifi.resume()
- node.sleep()
- node.dsleepMax()

node.dsleepMax()

Повертає поточну теоретичну максимальну тривалість глибокого сну.

Синтаксис

node.dsleepMax()

Параметри

немає

Повернення

max_duration

приклад

```
node.dsleep(node.dsleepMax())
```

Дивись також

- node.dsleep()

node.flashid()

Повертає ідентифікатор флеш-чіпа.

Синтаксис

```
node.flashid()
```

Параметри

немає

Повернення

flash ID (номер)

node.flashsize()

Повертає розмір мікросхеми флеш-пам’яті в байтах. На модулях 4 МБ, таких як ESP-12, повертається значення 4194304 = 4096 КБ.

Синтаксис

```
node.flashsize()
```

Параметри

немає

Повернення

розмір флеш-пам'яті в байтах (ціле число)

node.getcpufreq()

Отримує поточну частоту ЦП.

Синтаксис

```
node.getcpufreq()
```

Параметри

немає

Повернення

Поточна частота ЦП (число)

приклад

```
do
  local cpuFreq = node.getcpufreq()
  print("The current CPU frequency is " .. cpuFreq .. " MHz")
end
```

node.getpartitiontable()

Отримує поточну інформацію про розділи LFS і SPIFFS.

Синтаксис

`node.getpartitiontable()`

Параметри

немає

Повернення

Масив, що містить записи для `lfs_addr`, і . Значення адреси є зсувами відносно початку флеш-пам'яті. `lfs_sizespiffs_addrspiffs_size`

приклад

```
print("The LFS size is " .. node.getpartitiontable().lfs_size)
```

Дивись також

`node.setpartitiontable()`

node.heap()

Повертає поточний доступний розмір купи в байтах. Зауважте, що через фрагментацію фактичні розподіли такого розміру можуть бути неможливими.

Синтаксис

`node.heap()`

Параметри

немає

Повернення

залишився розмір системної купи в байтах (число)

node.info()

Повертає інформацію про апаратне забезпечення, версію програмного забезпечення та конфігурацію збірки.

Синтаксис

`node.info([group])`

Параметри

`group` Позначник для групи властивостей. Може бути одним із `"hw"`, `"sw_version"`, `"build_config"`.

Повернення

Якщо `group` задано а, повертається значення таблиці, що містить такі елементи:

- для `group` = `"hw"`
  - `chip_id` (число)
  - `flash_id` (число)
  - `flash_size` (число)
  - `flash_mode` (число) 0 = QIO, 1 = QOUT, 2 = DIO, 15 = DOUT.
  - `flash_speed` (число)
- для `group` = `"lfs"`
  - `lfs_base` (число) Зсув спалаху вибраної області LFS
  - `lfs_mapped` (число) Відображена адреса пам'яті вибраного регіону LFS
  - `lfs_size` (число) розмір вибраної області LFS
  - `lfs_used` (число) фактичний розмір, який використовується поточним зображенням LFS
- для `group` = `"sw_version"`

- `git_branch` (рядок)
- `git_commit_id` (рядок)
- `git_release` (рядок) назва випуску + додаткові коміти, наприклад, "2.0.0-master_20170202 +403"
- `git_commit_dts` (рядок) зафіксувати позначку часу у форматі впорядкування. наприклад, "201908111200"
- `node_version_major` (число)
- `node_version_minor` (число)
- `node_version_revision` (число)

- для `group` = `"build_config"`
  - `ssl` (логічне значення)
  - `lfs_size` (число), як визначено під час створення
  - `modules` (рядок) розділений комами список
  - `number_type` (рядок) `integer` або `float`
- для `group` = `nil`
  - `majorVer` (число)
  - `minorVer` (число)
  - `devVer` (число)
  - `chipid` (число)
  - `flashid` (число)
  - `flashsize` (число)
  - `flashmode` (число)
  - `flashspeed` (число)

node.input()

Надсилає рядок інтерпретатору Lua. Подібно до pcall(loadstring(str)), але без обмеження в один рядок. Зауважте, що дії інтерпретатора Line завершують лише фрагменти Lua. Фрагмент Lue Lua повинен складатися з одного або кількох '\n' закінчених рядків, які утворюють повну одиницю компіляції.

Синтаксис

node.input(str)

Параметри

str Lua-фрагмент

Повернення

nil

приклад

```
sk:on("receive", function(conn, payload) node.input(payload) end)
```

Перегляньте робочий приклад модуля Telnet Lua .

Дивись також

node.output()

node.LFS

Підтаблиця, що містить API для доступу до Lua Flash Store (**LFS**). Програмісти можуть віддати перевагу зіставити це з глобальною або локальною змінною для зручності, наприклад:

```
local LFS = node.LFS
```

ОПИС ФУНКЦІЙ API МОДУЛЯ NODE

node.LFS.get()

Повертає посилання на функцію для функції в LFS.

Зауважте, що невикористані `node.LFS` властивості відображаються на еквівалентному `get()` виклику, наприклад: `node.LFS.mySub1` є синонімом для `node.LFS.get('mySub1')`.

Синтаксис

`node.LFS.get(modulename)`

Параметри

`modulename` Назва модуля, який потрібно завантажити.

Повернення

- Якщо LFS завантажується, а це `modulename` рядок, який є назвою дійсного модуля в LFS, тоді функція повертається таким же чином, як `load()` і інші функції завантаження Lua
- В іншому випадку `nil` повертається.

node.LFS.list()

Перелічіть модулі в LFS.

Повернення

- Якщо зображення LFS НЕ ЗАВАНТАЖЕНО, `nil` повертається.
- В іншому випадку повертається відсортований масив імен модулів у LFS.

node.LFS.reload()

Перезавантажте LFS із наданим флеш-образом. Flash-зображення можна створити на головній машині за допомогою luac.cross команди.

Синтаксис

node.LFS.reload(imageName)

Параметри

imageName Ім'я файлу зображення у файловій системі, який буде завантажено в LFS.

Повернення

- У випадку, коли imagename це дійсний образ LFS, він розгортається та завантажується у флеш-пам'ять, а ESP негайно перезавантажується, *тому керування не повертається до програми Lua, що викликає,* у разі успішного перезавантаження.
- Внутрішній процес перезавантаження виконує кілька проходів через файл зображення LFS. Перший прохід перевіряє формати файлів і заголовків і виявляє багато помилок. Якщо будь-який виявлений, повертається рядок помилки.

node.output()

Перенаправляє інтерпретатор Lua до stdout каналу, коли вказано функцію CB (див. pipe модуль), і скидає вихід до нормального в іншому випадку. За бажанням також друкує на послідовну консоль.

Синтаксис

node.output(function(pipe), serial_debug)

Параметри

- `output_fn(pipe)` функція приймає кожен вихід як str і може надсилати вихід у сокет (або, можливо, у файл). Зауважте, що ця функція має відповідати правилам зворотного виклику зчитувача труб.
- `serial_debug` 1 вихід також відображається в послідовному режимі. 0: немає послідовного виведення.

Повернення

`nil`

приклад

Перегляньте робочий приклад модуля Telnet Lua .

Дивись також

`node.input()`

node.restart()

Перезапускає чіп.

Синтаксис

`node.restart()`

Параметри

немає

Повернення

`nil`

node.restore()

Відновлює конфігурацію системи до стандартних значень за допомогою функції SDK `system_restore()`, яка описана в документації як:

Скидання налаштувань за замовчуванням таких API: `wifi_station_set_auto_connect`, `wifi_set_phy_mode`, `wifi_softap_set_config`

пов'язаних, `wifi_station_set_config` пов'язаних `wifi_set_opmode`, і інформації про точки доступу, записану `#define AP_CACHE`.

Синтаксис

`node.restore()`

Параметри

немає

Повернення

`nil`

приклад

```
node.restore()
node.restart() -- ensure the restored settings take effect
```

node.setcpufreq()

Змініть робочу частоту ЦП.

Синтаксис

`node.setcpufreq(speed)`

Параметри

`speed` константа «node.CPU80MHZ» або «node.CPU160MHZ»

Повернення

цільова частота ЦП (число)

приклад

```
node.setcpufreq(node.CPU80MHZ)
```

node.setonerror()

Замінює стандартну обробку збоїв, яка завжди перезавантажує систему. Його можна використовувати, наприклад, щоб записати повідомлення про помилку в журнал або захистити підключене обладнання перед перезапуском.

Увага

Настійно рекомендується переконатися, що зворотний виклик завершується перезапуском. Щось пішло не так, і, ймовірно, небезпечно просто чекати наступної події (наприклад, тикання таймера) і сподіватися, що все вийде.

Синтаксис

`node.setonerror(function)`

Параметри

`function` функція зворотного виклику, яка виконується, коли виникає помилка, отримує рядок помилки як аргумент, не забудьте запустити **перезапуск** наприкінці зворотного виклику

Повернення

`nil`

приклад

```
node.setonerror(function(s)
    print("Error: "..s)
    node.restart()
  end)
```

node.setpartitiontable()

Встановлює поточну інформацію про розділ LFS та/або SPIFFS.

Синтаксис

`node.setpartitiontable(partition_info)`

Параметри

Масив, що містить одну або декілька з наведених нижче сутностей. Значення адреси є зсувом байтів відносно початку флеш-пам'яті. Значення розміру в байтах. Зауважте, що ці параметри мають бути кратними 8 Кб, щоб вирівняти межі сторінки Flash. - `lfs_addr`. Базова адреса регіону ОРС. - `lfs_size`. Розмір регіону LFS. - `spiffs_addr`. Базова адреса регіону SPIFFS. - `spiffs_size`. Розмір області SPIFFS.

Повернення

Не застосовується. Модуль ESP буде перезавантажено для нового дійсного набору або буде видано помилку Lua, якщо будуть виявлені невідповідності.

приклад

```
node.setpartitiontable{lfs_size = 0x20000, spiffs_addr = 0x120000,
spiffs_size = 0x20000}
```

Дивись також

node.getpartitiontable()

node.sleep()

Переведіть NodeMCU в режим легкого сну, щоб зменшити споживання струму.

- NodeMCU не може перейти в легкий сплячий режим, якщо Wi-Fi призупинено.
- Усі активні таймери буде призупинено, а потім відновлено, коли NodeMCU виходить із режиму сну.

Синтаксис

node.sleep({wake_pin[, int_type, resume_cb, preserve_mode]})

Параметри

- wake_pin 1-12, контакт, до якого потрібно приєднати переривання пробудження. Зауважте, що контакт 0 (GPIO 16) не підтримує переривання.
  - GPIO module Для отримання додаткової інформації про пін-карту зверніться до .
- int_type тип переривання, яке ви хочете прокинути. (Необов'язково, за замовчуванням: node.INT_LOW)
  - допустимі режими переривання:
    - node.INT_UP Висхідний край

- node.INT_DOWN Падаючий край
- node.INT_BOTH Обидва краю
- node.INT_LOW Низький рівень
- node.INT_HIGH Високий рівень

resume_cb Зворотний виклик для виконання, коли WiFi виходить із режиму призупинення. (необов'язково)

preserve_mode зберегти поточний режим WiFi через сплячий режим вузла. (Необов’язково, за замовчуванням: true)

- Якщо значення true, режими Station і StationAP автоматично відновлять підключення до попередньо налаштованої точки доступу, коли NodeMCU відновить роботу.
- Якщо false, відмініть режим WiFi і залиште NodeMCU у wifi.NULL_MODE. Режим Wi-Fi буде відновлено до вихідного режиму після перезавантаження.

Повернення

- nil

приклад

```
--Put NodeMCU in light sleep mode indefinitely with resume callback and
wake interrupt
 cfg={}
 cfg.wake_pin=3
 cfg.resume_cb=function() print("WiFi resume") end

 node.sleep(cfg)

--Put NodeMCU in light sleep mode with interrupt, resume callback and
discard WiFi mode
 cfg={}
 cfg.wake_pin=3 --GPIO0
 cfg.resume_cb=function() print("WiFi resume") end
 cfg.preserve_mode=false

 node.sleep(cfg)
```

Дивись також

- wifi.suspend()
- wifi.resume()
- node.dsleep()

node.startupcommand()

Замінює дію запуску за замовчуванням під час перезавантаження процесора, замінюючи виконання, init.lua якщо воно існує. Тепер це застаріло на користь node.startup({command="the command"}).

Синтаксис

node.startupcommand(string)

Параметри

- string з префіксом або
  - @, рядок, що залишився, є ім'ям файлу, який потрібно виконати.
  - =, рядок, що залишився, є фрагментом Lua, який потрібно скомпілювати та виконати.

Повернення

```
`status` this is `false` if write to the Reboot Config Record fails.  Note
that no attempt is made to   parse or validate the string. If the command
is invalid or the file missing then this will be reported on the next
restart.
```

приклад

```
node.startupcommand("@myappstart.lc")  -- Execute the compiled file
myappstart.lc on startup
-- Execute the LFS routine init() in preference to init.lua
node.startupcommand("=if LFS.init then LFS.init() else dofile('init.lua')
end")
```

node.startupcounts()

Запитати продуктивність запуску системи.

важливо

Ця функція доступна, лише якщо мікропрограму зібрано з PLATFORM_STARTUP_COUNT визначеним. Зазвичай це можна зробити, розкоментувавши #define PLATFORM_STARTUP_COUNT рядок у app/include/user_config.h.

Синтаксис

`node.startupcounts([marker])`

Параметри

- `marker` Якщо є, це додасть ще один запис до підрахунків запуску

Повернення

Масив таблиць, які вказують, скільки циклів процесора було використано на кожному кроці завантаження платформи.

приклад

```
=sjson.encode(node.startupcounts())
```

Це може створити вихід (відформатований для зручності читання):

```
[
 {"ccount":3774328,"name":"user_pre_init","line":124},
 {"ccount":3842297,"name":"user_pre_init","line":180},
 {"ccount":9849869,"name":"user_init","line":327},
 {"ccount":10008843,"name":"nodemcu_init","line":293},
 {"ccount":10295779,"name":"pmain","line":234},
 {"ccount":11378766,"name":"pmain","line":256},
 {"ccount":11565912,"name":"pmain","line":260},
 {"ccount":12158242,"name":"node_startup_counts","line":1},
 {"ccount":12425790,"name":"myspiffs_mount","line":126},
 {"ccount":12741862,"name":"myspiffs_mount","line":148},
 {"ccount":13983567,"name":"pmain","line":265}
]
```

Вирішальним записом є запис, для `node_startup_counts` якого запущена програма. Це було на Wemos D1 Mini зі спалахом, що працює на частоті 80 МГц. Усі параметри запуску були ввімкнені. Зверніть увагу, що тактова частота змінюється `user_pre_init` до 160 МГц. Загальний час становив

(приблизно): `3.8 / 80 + (12 - 3.8) / 160 = 98ms`. З параметрами запуску 0 час становить 166 мс. Цей час може бути трохи оптимістичним, оскільки годинник під час завантаження деякий час становить лише 52 МГц.

node.startup()

Отримати/встановити параметри, які керують процесом запуску. З часом цей інтерфейс буде розвиватися.

Синтаксис

`node.startup([{table}])`

Параметри

Якщо аргумент опущено, поточний набір параметрів запуску не змінюється. Якщо аргументом є порожня таблиця, `{}` усі параметри скидаються до значень за замовчуванням.

- `table` один або декілька варіантів:
  - `banner` - встановіть значення true або false, щоб вказати, чи має відображатися банер запуску чи ні. (за замовчуванням: true)
  - `frequency` - встановити node.CPU80MHZ або node.CPU160MHZ, щоб вказати початкову швидкість ЦП. (за замовчуванням: node.CPU80MHZ)
  - `delay_mount` - встановіть значення true або false, щоб вказати, чи відкладено монтування файлової системи SPIFFS до першої необхідності чи ні. (за замовчуванням: false)
  - `command` - встановити рядок, який є початковою командою, яка виконується. Це той самий рядок, що й у `node.startupcommand`.

Повернення

`table` Це повний набір параметрів у стані, який набуде чинності під час наступного завантаження. Зауважте, що `command` ключ може бути відсутнім - у цьому випадку буде використано значення за замовчуванням.

приклад

```
node.startup({banner=false, frequency=node.CPU160MHZ})  -- Prevent
printing the banner and run at 160MHz
```

node.stripdebug()

Контролює обсяг налагоджувальної інформації, що зберігається протягом `node.compile()`, і дозволяє видаляти налагоджувальну інформацію з уже скомпільованого коду Lua.

Рекомендовано лише для досвідчених користувачів, стандартні параметри NodeMCU підходять майже для всіх випадків використання.

Синтаксис

`node.stripdebug([level[, function]])`

Параметри

- `level`
  - 1, не відкидати інформацію про налагодження
  - 2, відкидати інформацію про налагодження Local і Upvalue
  - 3, відкидати інформацію про налагодження Local, Upvalue і номер рядка
- `function` скомпільована функція, яку потрібно видалити для setfenv, крім 0,

Якщо аргументи не надано, повертається поточне налаштування за замовчуванням. Якщо функція пропущена, це налаштування за замовчуванням для майбутніх компіляцій. Аргумент функції використовує ті самі правила, що й для `setfenv()`.

Повернення

Якщо викликати без аргументів, повертає налаштування поточного рівня.

В іншому випадку `nil` повертається.

приклад

```
node.stripdebug(3)
node.compile('bigstuff.lua')
```

Дивись також

`node.compile()`

node.osprint()

Контролює, чи друкуються результати налагодження з Espressif SDK. Зауважте, що це доступно, лише якщо вбудоване програмне забезпечення створено з визначеними DEVELOPMENT_TOOLS.

Синтаксис

`node.osprint(enabled)`

Параметри

- `enabled` Це дозволяє або `true` ввімкнути друк, або `false` вимкнути його.

  Типовим є `false`.

Повернення

нічого

приклад

```
node.osprint(true)
```

node.random()

Це веде себе як math.random, за винятком того, що використовує справжні випадкові числа, отримані від обладнання ESP8266.

Він повертає рівномірно розподілені числа в потрібному діапазоні. Він також піклується про правильні великі діапазони.

Його можна назвати трьома способами. Без аргументів у збірці з плаваючою комою NodeMCU він повертає випадкове дійсне число з рівномірним розподілом в інтервалі [0,1).

При виклику лише з одним аргументом, цілим числом n, він повертає ціле випадкове число x таке, що 1 <= x <= n.

Наприклад, ви можете змоделювати результат кубика за допомогою random(6).

Нарешті, random можна викликати з двома цілочисельними аргументами, l і u, щоб отримати псевдовипадкове ціле x таке, що l <= x <= u.

Синтаксис

`node.random()` `node.random(n)` `node.random(l, u)`

Параметри

- `n` кількість різних цілих значень, які можна повернути -- у діапазоні (включно) 1 .. `n`
- `l` нижня межа діапазону
- `u` верхня межа діапазону

Повернення

Випадкове число у відповідному діапазоні. Зверніть увагу, що форма нульового аргументу завжди повертатиме 0 у цілочисельній збірці.

приклад

```
print ("I rolled a", node.random(6))
```

МОДУЛЬ NODE.EGC

node.egc.setmode()

Встановлює режим аварійного збирача сміття

Синтаксис

`node.egc.setmode(mode, [param])`

Параметри

- `mode`
  - `node.egc.NOT_ACTIVE` EGC неактивний, жодного циклу збору не буде примусово в ситуаціях із нестачею пам’яті
  - `node.egc.ON_ALLOC_FAILURE` Спробуйте виділити новий блок пам'яті та запустіть збирач сміття, якщо розподіл не вдається. Якщо розподіл не вдається навіть після запуску збирача сміття, розподільник повернеться з помилкою.
  - `node.egc.ON_MEM_LIMIT` Запустіть збирач сміття, коли обсяг пам’яті, який використовує сценарій Lua, перевищує верхнє значення `limit`. Якщо верхня межа не може бути задоволена навіть після запуску збирача сміття, розподільник повернеться з помилкою. Якщо даний ліміт є від’ємним, він інтерпретується як бажана кількість купи, яку слід залишити доступною. Щоразу, коли вільна купа (як повідомляється `node.heap()` опускається нижче запитаного ліміту, збирач сміття буде запущено.
  - `node.egc.ALWAYS` Запускайте збирач сміття перед кожним виділенням пам'яті. Якщо розподіл не вдається навіть після запуску збирача сміття, розподільник повернеться з помилкою. Цей режим дуже ефективний з точки зору економії пам'яті, але він також найповільніший.

- level у випадку node.egc.ON_MEM_LIMIT, це визначає обмеження пам'яті.

Повернення

nil

приклад

```
node.egc.setmode(node.egc.ALWAYS, 4096)
-- This is the default setting at
startup. node.egc.setmode(node.egc.ON_ALLOC_FAILURE)
-- This is the fastest activeEGC
mode. node.egc.setmode(node.egc.ON_MEM_LIMIT, 30720)
-- Only allow the Lua runtime to allocate at most 30k, collect garbage if
limit is about to be hit node.egc.setmode(node.egc.ON_MEM_LIMIT, -6144)
-- Try to keep at least 6k heap available for non-Lua use (e.g. network
buffers)
```

node.egc.meminfo()

Повертає інформацію про використання пам’яті для середовища виконання Lua.

Синтаксис

total_allocated, estimated_used = node.egc.meminfo()

Параметри

Жодного.

Повернення

- total_allocated Загальна кількість байтів, виділених середовищем виконання Lua. Це число, яке має значення при використанні node.egc.ON_MEM_LIMIT опції з позитивними граничними значеннями.
- estimated_used Це значення показує приблизне використання виділеної пам’яті.

МОДУЛЬ NODE.TASK

node.task.post()

Увімкає зворотний виклик Lua або завдання, щоб опублікувати інший запит завдання. Зауважте, що відповідно до прикладу в будь-якому завданні можна опублікувати кілька завдань, але найвищий пріоритет завжди доставляється першим.

Якщо черга завдань заповнена, виникає повідомлення про помилку queue full.

Синтаксис

`node.task.post([task_priority], function)`

Параметри

- `task_priority` (необов'язково)
  - `node.task.LOW_PRIORITY` = 0
  - `node.task.MEDIUM_PRIORITY` = 1
  - `node.task.HIGH_PRIORITY` = 2
- `function` функція зворотного виклику, яка буде виконана під час виконання завдання.

Якщо пріоритет опущено, за замовчуванням використовується значення `node.task.MEDIUM_PRIORITY`

Повернення

`nil`

приклад

```
for i = node.task.LOW_PRIORITY, node.task.HIGH_PRIORITY do
  node.task.post(i,function(p2)
    print("priority is "..p2)
  end)
end
priority is 2
priority is 1
priority is 0
```

МОДУЛЬ MQTT

Клієнт підтримує версію 3.1.1 протоколу MQTT . Переконайтеся, що ваш брокер підтримує та правильно налаштований для версії 3.1.1. Клієнт зворотно несумісний з брокерами, які працюють на MQTT 3.1.

mqtt.Client()	Створює клієнт MQTT.
mqtt.client:close()	Планує чистий розрив з'єднання.
mqtt.client:connect()	Підключається до посередника, указаного вказаним хостом, портом і параметрами безпеки.
mqtt.client:lwt()	Налаштування Last Will and Testament.
mqtt.client:on()	Реєструє функцію зворотного виклику для події.
mqtt.client:publish()	Публікує повідомлення.
mqtt.client:subscribe()	Підписується на одну або декілька тем.
mqtt.client:unsubscribe()	Відписується від однієї або кількох тем.

ОПИС ФУНКЦІЙ API МОДУЛЯ MQTT

mqtt.Client()

Створює клієнт MQTT.

Синтаксис

```
mqtt.Client(clientid, keepalive[, username, password, cleansession,
max_message_length])
```

Параметри

- `clientid` ідентифікатор клієнта
- `keepalive` підтримувати активність секунд
- `username` ім'я користувача

- password пароль користувача
- cleansession 0/1 для false / true. За замовчуванням 1 (true).
- max_message_length, наскільки великі повідомлення приймати. За замовчуванням 1024.

Повернення

Клієнт MQTT

Примітки

Відповідно до специфікації MQTT максимальна довжина PUBLISH становить 256 Мб.

Це занадто велике, щоб NodeMCU реалістично впорався. Щоб уникнути ситуації браку пам'яті, існує обмеження щодо розміру повідомлень, які потрібно приймати.

Це контролюється параметром max_message_length. На практиці це впливає лише на вхідні повідомлення PUBLISH, оскільки всі звичайні контрольні пакети малі.

Було вибрано значення за замовчуванням 1024, оскільки це було неявним обмеженням у NodeMCU 2.2.1 і старіших версіях (де це взагалі не оброблялося).

Зверніть увагу, що «довжина повідомлення» стосується повного розміру повідомлення MQTT, включаючи фіксовані та змінні заголовки, назву теми, ідентифікатор пакета (якщо застосовно) і корисне навантаження.

Будь-яке повідомлення, *розмір якого перевищує*, max_message_length буде (частково) доставлено до overflow зворотного виклику, якщо визначено.

Решту повідомлення буде видалено. Будь-які наступні повідомлення слід обробляти належним чином.

Відкинуті повідомлення все одно отримуватимуть підтвердження, якщо було подано запит на рівень QoS 1 або 2, навіть якщо стек програми не зможе їх обробити.

Пам'ять купи буде використовуватися для буферизації будь-якого повідомлення, яке охоплює більше ніж один TCP-пакет.

Єдиний розподіл для повного повідомлення буде виконано, коли заголовок повідомлення вперше побачено, щоб уникнути фрагментації купи. Якщо розподіл не вдається, сеанс MQTT буде відключено.

Природно, повідомлення, розмір яких перевищує , max_message_length не зберігатимуться.

Зауважте, що виділення купи може відбутися, навіть якщо окремі повідомлення не перевищують налаштований максимум!

Наприклад, брокер може надіслати кілька менших повідомлень у швидкій послідовності, які можуть входити в той самий пакет TCP.

Якщо останнє повідомлення в TCP-пакеті не вміщується повністю, буде виділено буфер купи для зберігання неповного повідомлення під час очікування наступного TCP-пакета.

Типовий максимальний розмір повідомлення, яке вміщується в один пакет TCP, становить 1460 байт, але це залежить від конфігурації MTU мережі, фрагментації пакетів і, як описано вище, кількох повідомлень в одному пакеті TCP.

приклад

```
-- init mqtt client without logins, keepalive timer 120s
m = mqtt.Client("clientid", 120)

-- init mqtt client with logins, keepalive timer 120sec
m = mqtt.Client("clientid", 120, "user", "password")

-- setup Last Will and Testament (optional)
-- Broker will publish a message with qos = 0, retain = 0, data = 
"offline"
-- to topic "/lwt" if client don't send keepalive packet
m:lwt("/lwt", "offline", 0, 0)

m:on("offline", function(client) print ("offline") end)

-- on publish message receive event
m:on("message", function(client, topic, data)
  print(topic .. ":" )
  if data ~= nil then
```

```
    print(data)
  end
end)

-- on publish overflow receive event
m:on("overflow", function(client, topic, data)
  print(topic .. " partial overflowed message: " .. data )
end)

-- for TLS: m:connect("192.168.11.118", secure-port, 1)
m:connect("192.168.11.118", 1883, false, function(client)
  print("connected")
  -- Calling subscribe/publish only makes sense once the connection
  -- was successfully established. You can do that either here in the
  -- 'connect' callback or you need to otherwise make sure the
  -- connection was established (e.g. tracking connection status or in
  -- m:on("connect", function)).

  -- subscribe topic with qos = 0
  client:subscribe("/topic", 0, function(client) print("subscribe
success") end)
  -- publish a message with data = hello, QoS = 0, retain = 0
  client:publish("/topic", "hello", 0, 0, function(client) print("sent")
end)
end,
function(client, reason)
  print("Connection failed reason: " .. reason)
end)

m:close()
-- you can call m:connect again after the offline callback fires
```

mqtt.client:close()

Планує чистий розрив з’єднання.

MQTT вимагає від клієнтів активно сигналізувати про бажання відключитися від сервера, щоб уникнути надсилання свого LWT. Таким чином, Клієнт можна повторно використовувати не одразу після цього виклику, а лише після того, як спрацював зворотний виклик «офлайн».

Синтаксис

```
mqtt:close()
```

Параметри

немає

Повернення

`nil`

mqtt.client:connect()

Підключається до посередника, указаного вказаним хостом, портом і параметрами безпеки.

Синтаксис

`mqtt:connect(host[, port[, secure]][, function(client)[, function(client, reason)]])`

Параметри

- `host` хост, домен або IP (рядок)
- `port` порт брокера (номер), за замовчуванням 1883
- `secure` логічний: якщо `true`, використовувати TLS. Зверніть увагу на обмеження, задокументовані в модулі net .
- `function(client)` функція зворотного виклику, коли з'єднання встановлено
- `function(client, reason)` функція зворотного виклику, коли з’єднання не вдається встановити. Подальші зворотні виклики не повинні бути викликані.

Повернення

`nil`; використовуйте зворотні виклики, щоб спостерігати за результатом.

Примітки

Програма повинна стежити за збоями підключення та обробляти помилки зворотного виклику помилки, щоб досягти надійного підключення до сервера.

Наприклад:

```
function handle_mqtt_error(client, reason)
  tmr.create():alarm(10 * 1000, tmr.ALARM_SINGLE, do_mqtt_connect)
end

function do_mqtt_connect()
```

```
  mqtt:connect("server", function(client) print("connected") end,
handle_mqtt_error)
end
```

Перший зворотний виклик до :connect() псевдонімів із доступним зворотним викликом "connect" :on() (використовується останній переданий зворотний виклик до будь-якого з них).

Однак, якщо nil передати в :connect(), будь-який існуючий зворотний виклик буде збережено, а не видалено.

Другий (збій) зворотний виклик псевдонімів із зворотним викликом "connfail", доступним через :on(). (Зворотний виклик «офлайн» викликається лише після того, як уже встановлене з'єднання стає закритим.

Якщо під час connect() виклику не вдається встановити з'єднання, :connect() викликається зворотний виклик, який передається, і нічого більше.)

Коди причин зворотного виклику збою підключення:

Постійний	**Значення**	**опис**
mqtt.CONN_FAIL_SERVER_NOT_FOUND	-5	Немає брокера, який прослуховує вказану IP-адресу та порт
mqtt.CONN_FAIL_NOT_A_CONNACK_MSG	-4	Відповідь брокера не була CONNACK, як того вимагає протокол
mqtt.CONN_FAIL_DNS	-3	Помилка пошуку DNS
mqtt.CONN_FAIL_TIMEOUT_RECEIVING	-2	Час очікування CONNACK від брокера

Постійний	Значення	опис
mqtt.CONN_FAIL_TIMEOUT_SENDING	-1	Час очікування під час спроби надіслати повідомлення Connect
mqtt.CONNACK_ACCEPTED	0	Помилок немає. *Примітка. Це не призведе до зворотного виклику помилки.*
mqtt.CONNACK_REFUSED_PROTOCOL_VER	1	Брокер не є брокером 3.1.1 MQTT.
mqtt.CONNACK_REFUSED_ID_REJECTED	2	Зазначений ClientID було відхилено брокером. (Побачити `mqtt.Client()`)
mqtt.CONNACK_REFUSED_SERVER_UNAVAILABLE	3	Сервер недоступний.
mqtt.CONNACK_REFUSED_BAD_USER_OR_PASS	4	Брокер відмовився від вказаного логіна або пароля.
mqtt.CONNACK_REFUSED_NOT_AUTHORIZED	5	Ім'я користувача не авторизоване.

mqtt.client:lwt()

Налаштування Last Will and Testament .

Оскільки остання воля надсилається брокеру при підключенні, `lwt()` потрібно дзвонити ДО дзвінка `connect()`.

Брокер опублікує останнє повідомлення клієнта, як тільки помітить, що з’єднання з клієнтом розірвано; це відбувається, коли... –

Клієнт не може надіслати пакет підтримки активності протягом зазначеного часу mqtt.Client() - TCP-з'єднання належним чином закрито (без попереднього закриття mqtt-з'єднання) - Посередник намагається надіслати дані клієнту та TCP розриви зв'язку.

Це означає, що якщо ви вказали 120 як таймер підтримки активності, просто вимкніть клієнтський пристрій і брокер не надсилатиме жодних даних клієнту, останнє повідомлення буде опубліковано через 120 с після вимкнення пристрою.

Синтаксис

`mqtt:lwt(topic, message[, qos[, retain]])`

Параметри

- `topic` тема для публікації (рядок)
- `message` повідомлення для публікації (буфер або рядок)
- `qos` Рівень QoS, за умовчанням 0
- `retain` зберегти прапор, за умовчанням 0

Повернення

`nil`

mqtt.client:on()

Реєструє функцію зворотного виклику для події.

Синтаксис

`mqtt:on(event, function(client[, topic[, message]]))`

Параметри

- `event` може бути "connect", "connfail", "suback", "unsuback", "puback", "message", "overflow" або "offline"

- функція зворотного виклику. Першим параметром завжди є сам клієнтський об’єкт. Решта переданих параметрів відрізняються залежно від події:
- Якщо подією є «повідомлення», 2-й і 3-й параметри отримують тему і повідомлення відповідно як рядки Lua.
- Якщо подія "переповнення", параметри такі ж, як і для "повідомлення", за винятком того, що рядок повідомлення скорочується до максимального розміру повідомлення.
- Якщо подією є "connfail", другим параметром буде код помилки підключення; Дивись вище.
- Інші типи подій не надають додаткових аргументів. Це має деякі сумні наслідки: інформація про максимальну якість обслуговування, надану брокером для підписки, втрачається, і програма повинна сама керувати чергою чи чергами, якщо вона очікує підтвердження подій.

Повернення

`nil`

mqtt.client:publish()

Публікує повідомлення.

Синтаксис

```
mqtt:publish(topic, payload, qos, retain[, function(client)])
```

Параметри

- `topic` тема для публікації (рядок теми)
- `message` повідомлення для публікації (буфер або рядок)
- `qos` Рівень QoS
- `retain` зберегти прапор

- `function(client)` додатковий зворотний виклик, що запускається, коли PUBACK отримано (для QoS 1 або 2) або коли повідомлення надіслано (для QoS 0).

Примітки

Під час виклику publish() кілька разів для ВСІХ команд публікації буде викликана остання визначена функція зворотного виклику. Цей аргумент зворотного виклику також є псевдонімом зворотного виклику "puback" для `:on()`.

Повернення

`true` на успіх, `false` інакше

mqtt.client:subscribe()

Підписується на одну або декілька тем.

Синтаксис

`mqtt:subscribe(topic, qos[, function(client)])` `mqtt:subscribe(table[, function(client)])`

Параметри

- `topic` рядок теми
- `qos` Рівень підписки на QoS, за умовчанням 0
- `table` масив пар «тема, qos», на які можна підписатися
- `function(client)` необов'язковий зворотний виклик, що запускається, коли підписка(и) успішно виконана.

Примітки

Під час виклику subscribe() більше ніж один раз, остання визначена функція зворотного виклику буде викликана для ВСІХ команд підписки. Цей

аргумент зворотного виклику також пов'язаний із зворотним викликом «suback» для :on() .

Повернення

true на успіх, false інакше

приклад

```
-- subscribe topic with qos = 0
m:subscribe("/topic",0, function(conn) print("subscribe success") end)

-- or subscribe multiple topic (topic/0, qos = 0; topic/1, qos = 1; topic2
, qos = 2)
m:subscribe({["topic/0"]=0,["topic/1"]=1,topic2=2}, function(conn)
print("subscribe success") end)
```

Обережно

Замість того, щоб викликати subscribeкілька разів, вам слід використовувати синтаксис кількох тем, показаний у прикладі вище, якщо ви хочете підписатися на декілька тем одночасно.

mqtt.client:unsubscribe()

Відписується від однієї або кількох тем.

Синтаксис

mqtt:unsubscribe(topic[, function(client)]) mqtt:unsubscribe(table[, function(client)])

Параметри

- topic рядок теми
- table масив пар «тема, будь-що», від яких можна скасувати підписку
- function(client) необов'язковий зворотний виклик, що запускається, коли скасування підписки вдалось.

Примітки

Під час виклику subscribe() більше ніж один раз, остання визначена функція зворотного виклику буде викликана для ВСІХ команд підписки.

Цей аргумент зворотного виклику також є псевдонімом зворотного виклику "unsuback" для `:on()`.

Повернення

`true` на успіх, `false` інакше

приклад

```
-- unsubscribe topic
m:unsubscribe("/topic", function(conn) print("unsubscribe success") end)

-- or unsubscribe multiple topic (topic/0; topic/1; topic2)
m:unsubscribe({["topic/0"]=0,["topic/1"]=0,topic2="anything"},
function(conn) print("unsubscribe success")
```

МОДУЛЬ SJSON

Модуль підтримки JSON. Дозволяє кодувати та декодувати в/з JSON.

Зауважте, що для кодування вкладених таблиць може знадобитися багато пам’яті. Щоб виявити помилки нестачі пам’яті, використовуйте `pcall()`.

Модуль можна використовувати двома способами.

Простіший спосіб - використовувати його як пряме завантаження для cjson (зробити `_G.cjson = sjson`).

Більш просунутий підхід полягає у використанні потокового інтерфейсу. Це дозволяє кодувати та декодувати значно більші об’єкти.

Обробка json null виглядає наступним чином:

- За замовчуванням декодер представляє null як sjson.NULL (який є об’єктом userdata). Це поведінка cjson.
- Кодер завжди перетворює будь-який об’єкт userdata на null.
- За бажанням можна вказати один рядок і в кодувальнику, і в декодері. Цей рядок використовуватиметься під час кодування/декодування для представлення нульових значень json. Цей рядок не слід використовувати більше ніде у ваших структурах даних. Відповідним значенням може бути `"\0"`.

Під час кодування об’єкта Lua, якщо функція знайдена, вона викликається (без аргументів), а (єдине) повернуте значення кодується замість функції.

Примітка

Усі приклади нижче використовують JSON у пам’яті або вміст, зчитаний із файлової системи SPIFFS. Однак реалізація потокової передачі справді найкраща - це отримання великих структур JSON із віддалених ресурсів і вилучення значень на льоту. Розгорнутий приклад потокової передачі можна знайти в /lua_examplesпапці.

sjson.encoder()	Це створює об’єкт кодувальника, який може перетворювати об’єкт Lua на рядок у кодуванні JSON.
sjson.encoder: читання	Це отримує частину даних у кодуванні JSON.
sjson.encode()	Закодуйте таблицю Lua у рядок JSON.
sjson.decoder()	Це робить об’єкт декодера, який може аналізувати рядок у кодуванні JSON на об’єкт Lua.
sjson.decoder:write	Це надає більше даних для аналізу в об’єкт Lua.
sjson.decoder:результат	Це отримує декодований об’єкт Lua або викликає помилку, якщо декодування ще не завершено.
sjson.decode()	Декодуйте рядок JSON у таблицю Lua.
Константи	Є одна константа, sjson.

ОПИС ФУНКЦІЙ API МОДУЛЯ SJSON

sjson.encoder()

Створює об’єкт кодувальника, який може перетворювати об’єкт Lua на рядок у кодуванні JSON.

Синтаксис

```
sjson.encoder(table [, opts])
```

Параметри

- table дані для кодування
- opts додаткова таблиця параметрів. Можливі записи:
- depth максимальна глибина кодування, необхідна для кодування таблиці. За замовчуванням 20, чого має бути достатньо майже для всіх ситуацій.
- null значення рядка, яке слід вважати нульовим.

Повернення

Об'єкт `sjson.encoder`.

sjson.encoder: читання

Це отримує частину даних у кодуванні JSON.

Синтаксис

`encoder:read([size])`

Параметри

- `size` необов'язкове значення для кількості байтів для повернення. За замовчуванням 1024.

Повернення

Рядок довжиною до `size` байтів або `nil` якщо кодування завершено й усі дані повернуто.

приклад

У наступному прикладі друкується (64-байтовими фрагментами) рядок у кодуванні JSON, що містить перші 4 Кб кожного файлу у файловій системі. Загальний рядок може бути більшим, ніж загальний обсяг пам’яті на NodeMCU.

```
function files()
   result = {}
   for k,v in pairs(file.list()) do
     result[k] = function() return file.open(k):read(4096) end
   end
   return result
end

local encoder = sjson.encoder(files())

while true do
   data = encoder:read(64)
   if not data then
      break
   end
   print(data)
end
```

sjson.encode()

Закодуйте таблицю Lua у рядок JSON. Це зручний метод для зворотної сумісності з `cjson`.

Синтаксис

`sjson.encode(table [, opts])`

Параметри

- `table` дані для кодування
- `opts` додаткова таблиця параметрів. Можливі записи:
  - `depth` максимальна глибина кодування, необхідна для кодування таблиці. За замовчуванням 20, чого має бути достатньо майже для всіх ситуацій.
  - `null` значення рядка, яке слід вважати нульовим.

Повернення

Рядок JSON

приклад

```
ok, json = pcall(sjson.encode, {key="value"})
if ok then
  print(json)
else
  print("failed to encode!")
end
```

sjson.decoder()

Це робить об'єкт декодера, який може аналізувати рядок у кодуванні JSON на об'єкт Lua. Метатаблицю можна вказати для всіх щойно створених таблиць Lua. Це дозволяє обробляти кожне значення, коли воно вставляється в кожну таблицю (шляхом реалізації методу `__newindex`).

Синтаксис

`sjson.decoder([opts])`

Параметри

- `opts` додаткова таблиця параметрів. Можливі записи:
  - `depth` максимальна глибина кодування, необхідна для кодування таблиці. За замовчуванням 20, чого має бути достатньо майже для всіх ситуацій.
  - `null` значення рядка, яке слід вважати нульовим.
  - `metatable` таблиця для використання як метатаблиці для всіх нових таблиць у повернутому об'єкті.

Повернення

Об'єкт `sjson.decoder`

Метатабельний

Є два основні методи, які викликаються в метатаблиці (якщо вона присутня).

- `__newindex` це стандартний метод, який викликається кожного разу, коли створюється новий елемент таблиці.
- `checkpath` це викликається (якщо визначено) кожного разу, коли створюється нова таблиця. Він викликається з двома аргументами:
  - `table` це щойно створена таблиця
  - `path` це список ключів від кореня. Він повинен повертатися `true`, якщо цей об'єкт є потрібним у результаті або `false` в іншому випадку.

Наприклад, під час декодування `{ "foo": [1, 2, []] }` контрольний шлях буде викликано наступним чином:

- checkpath({}, {}) аргумент table - це об'єкт, який відповідатиме значенню об'єкта JSON.
- checkpath({}, {"foo"}) аргумент table - це об'єкт, який відповідатиме значенню зовнішнього масиву JSON.
- checkpath({}, {"foo", 3}) аргумент table - це об'єкт, який відповідатиме порожньому внутрішньому масиву JSON.

Під час checkpath виклику методу метатаблиця вже пов'язана з новою таблицею. Таким чином, checkpath метод може замінити його за бажанням. Наприклад, якщо ви декодуєте { "foo": { "bar": [1,2,3,4], "cat": [5] } } і з якоїсь причини не хочете захоплювати значення ключа "bar", то є різні способи зробити це:

- У __newindex метаметоді просто перевірте значення ключа та пропустіть, rawset якщо ключ "bar". Це працює, лише якщо ви хочете пропустити всі клавіші "bar".
- У checkpath методі, якщо шлях дорівнює ["foo"], тоді повертається false.
- Використовуйте наступне checkpath: checkpath=function(tab, path) tab['__json_path'] = path return true end це збереже шлях у кожному створеному об'єкті. Тепер __newindex метод може виконувати більш складну фільтрацію.

Причина можливості фільтрації полягає в тому, що вона дозволяє обробляти дуже великі відповіді JSON на платформі з обмеженим обсягом пам'яті. Багато API повертають багато інформації, яка перевищить бюджет пам'яті платформи.

sjson.decoder:write

Це надає більше даних для аналізу в об’єкт Lua.

Синтаксис

`decoder:write(string)`

Параметри

- `string` наступний фрагмент даних у кодуванні JSON

Повернення

Сконструйований об’єкт Lua або `nil` якщо декодування ще не завершено.

Помилки

Якщо під час цього декодування виникає помилка синтаксичного аналізу, видається помилка, і аналіз припиняється. Об'єкт не можна використовувати повторно.

sjson.decoder:результат

Це отримує декодований об’єкт Lua або викликає помилку, якщо декодування ще не завершено. Це можна викликати кілька разів і кожен раз повертатиме той самий об’єкт.

Синтаксис

`decoder:result()`

Помилки

Якщо декодування не завершено, видається помилка.

приклад

```
local decoder = sjson.decoder()

decoder:write("[10, 1")
decoder:write("1")
decoder:write(", \"foo\"]")

for k,v in pairs(decoder:result()) do
   print (k, v)
end
```

Наступний приклад демонструє використання аргументу метатаблиці. У цьому випадку він просто друкує операції, але за бажанням може повністю приховати призначення.

```
local decoder = sjson.decoder({metatable=
        {__newindex=function(t,k,v) print("Setting '" .. k .. "' = '" ..
tostring(v) .."'")
rawset(t,k,v) end}})

decoder:write('[1, 2, {"foo":"bar"}]')
```

sjson.decode()

Декодує рядок JSON у таблицю Lua. Це зручний метод для зворотної сумісності з `cjson`.

Синтаксис

`sjson.decode(str[, opts])`

Параметри

- `str` Рядок JSON для декодування
- `opts` додаткова таблиця параметрів. Можливі записи:
  - `depth` максимальна глибина кодування, необхідна для кодування таблиці. За замовчуванням 20, чого має бути достатньо майже для всіх ситуацій.
  - `null` значення рядка, яке слід вважати нульовим.
  - `metatable` таблиця для використання як метатаблиці для всіх нових таблиць у повернутому об’єкті. Перегляньте розділ метатаблиці в описі `sjson.decoder()` вище.

Повернення

Представлення даних JSON у таблиці Lua

Помилки

Якщо рядок недійсний JSON, видається помилка.

приклад

```
t = sjson.decode('{"key":"value"}')
```

```
for k,v in pairs(t) do print(k,v) end
```

Константи

Існує одна константа `sjson.NULL`, яка використовується в структурах Lua для представлення присутності нуля JSON.

МОДУЛЬ HTTP

Базовий *клієнтський* модуль HTTP, який надає інтерфейс для виконання GET/POST/PUT/DELETE через HTTP(S), а також налаштованих запитів.

Через обмеження пам'яті ESP8266 підтримуваний розмір сторінки/основної частини обмежується доступною пам'яттю.

Якщо потрібні більші розміри сторінки/основного тексту є можливість використання `net.createConnection()` та потокової передачі даних.

Кожен метод запиту приймає зворотний виклик, який викликається після отримання відповіді від сервера.

Першим аргументом є код статусу, який є або звичайним кодом статусу HTTP, або -1 для позначення DNS, помилки підключення чи браку пам'яті, або тайм-ауту (наразі 60 секунд).

Для кожної операції можна надати спеціальні заголовки HTTP або замінити стандартні заголовки. За замовчуванням `Host` заголовок виводиться з URL-адреси та `User-Agent` має значення `ESP8266`.

Однак зауважте, що `Connection` заголовок *не можна* перевизначити! Завжди встановлено значення `close`.

Переспрямування HTTP (статус HTTP 300-308) автоматично відслідковуються до обмеження 20, щоб уникнути страшних циклів перенаправлення.

Під час виклику зворотного виклику йому передається код статусу HTTP, тіло, яке було отримано, і таблиця заголовків відповіді.

Усі назви заголовків написані малим регістром, щоб полегшити доступ. Якщо є кілька заголовків з однаковою назвою, повертається лише останній.

http.delete()	Виконує запит HTTP DELETE.
http.get()	Виконує запит HTTP GET.
http.post()	Виконує запит HTTP POST.
http.put()	Виконує запит HTTP PUT.
http.request()	Виконує спеціальний HTTP-запит для будь-якого методу HTTP.

ОПИС ФУНКЦІЙ API МОДУЛЯ

http.delete()

Виконує запит HTTP DELETE. Зауважте, що одночасні запити не підтримуються.

Синтаксис

```
http.delete(url, headers, body, callback)
```

Параметри

- `url` URL-адреса для отримання, включаючи префікс `http://` або `https://`
- `headers` Необов'язкові додаткові заголовки для додавання, *включаючи \r\n* ; може бути `nil`
- `body` Тіло для публікації; має бути вже закодовано у відповідному форматі, але може бути порожнім
- `callback` Функція зворотного виклику, яка викликається, коли відповідь отримана або сталася помилка; він викликається з аргументами `status_code`, `body` і `headers`. У разі помилки `status_code` встановлюється на -1.

Повернення

nil

приклад

```
http.delete('http://httpbin.org/delete',
  "",
  "",
  function(code, data)
    if (code < 0) then
      print("HTTP request failed")
    else
      print(code, data)
    end
  end)
```

http.get()

Виконує запит HTTP GET.

Синтаксис

http.get(url, headers, callback)

Параметри

- url URL-адреса для отримання, включаючи префікс http:// або https://
- headers Необов'язкові додаткові заголовки для додавання, *включаючи \r\n* ; може бути nil
- callback Функція зворотного виклику, яка викликається, коли відповідь отримана або сталася помилка; він викликається з аргументами status_code, body і headers. У разі помилки status_code встановлюється на -1.

Повернення

nil

приклад

```
http.get("http://httpbin.org/ip", nil, function(code, data)
    if (code < 0) then
      print("HTTP request failed")
    else
```

```
    print(code, data)
  end
end)
```

http.post()

Виконує запит HTTP POST.

Синтаксис

`http.post(url, headers, body, callback)`

Параметри

- `url` URL-адреса для отримання, включаючи префікс `http://` або `https://`
- `headers` Необов'язкові додаткові заголовки для додавання, *включаючи \r\n* ; може бути `nil`
- `body` Тіло для публікації; має бути вже закодовано у відповідному форматі, але може бути порожнім
- `callback` Функція зворотного виклику, яка викликається, коли відповідь отримана або сталася помилка; він викликається з аргументами `status_code`, `body` і `headers`. У разі помилки `status_code` встановлюється на -1.

Повернення

`nil`

приклад

```
http.post('http://httpbin.org/post',
  'Content-Type: application/json\r\n',
  '{"hello":"world"}',
  function(code, data)
    if (code < 0) then
      print("HTTP request failed")
    else
      print(code, data)
    end
  end)
```

http.put()

Виконує запит HTTP PUT.

Синтаксис

`http.put(url, headers, body, callback)`

Параметри

- `url` URL-адреса для отримання, включаючи префікс `http://` або `https://`
- `headers` Необов'язкові додаткові заголовки для додавання, *включаючи \r\n* ; може бути `nil`
- `body` Тіло для публікації; має бути вже закодовано у відповідному форматі, але може бути порожнім
- `callback` Функція зворотного виклику, яка викликається, коли відповідь отримана або сталася помилка; він викликається з аргументами `status_code`, `body` і `headers`. У разі помилки `status_code` встановлюється на -1.

Повернення

`nil`

приклад

```
http.put('http://httpbin.org/put',
  'Content-Type: text/plain\r\n',
  'Hello!\nStay a while, and listen...\n',
  function(code, data)
    if (code < 0) then
      print("HTTP request failed")
    else
      print(code, data)
    end
  end)
```

http.request()

Виконує спеціальний HTTP-запит для будь-якого методу HTTP.

Синтаксис

`http.request(url, method, headers, body, callback)`

Параметри

- `url` URL-адреса для отримання, включаючи префікс `http://` або `https://`
- `method` Метод HTTP для використання, наприклад, "GET", "HEAD", "OPTIONS" тощо
- `headers` Необов'язкові додаткові заголовки для додавання, *включаючи \r\n* ; може бути `nil`
- `body` Тіло для публікації; має бути вже закодовано у відповідному форматі, але може бути порожнім
- `callback` Функція зворотного виклику, яка викликається, коли відповідь отримана або сталася помилка; він викликається з аргументами `status_code`, `body` і `headers`. У разі помилки `status_code` встановлюється на -1.

Повернення

`nil`

приклад

```
http.request("http://httpbin.org", "HEAD", "", "",
  function(code, data)
    if (code < 0) then
      print("HTTP request failed")
    else
      print(code, data)
    end
  end)
```

МОДУЛЬ HTTPD

Цей модуль забезпечує інтерфейс до компонента веб-сервера Espressif .

Модуль httpd реалізує підтримку як обслуговування статичних файлів, так і генерації динамічного вмісту.

Для статичних файлів усі файли мають розташовуватися під загальним префіксом ("webroot") у (віртуальній) файловій системі.

Модулю байдуже, чи підтримує базова файлова система каталоги чи ні, тому файли можуть обслуговуватися з файлових систем SPIFFS, FAT або будь-якої іншої, яка може бути змонтована.

На відміну від стандартної поведінки веб-сервера Espressif, цей модуль обслуговує статичні файли на основі розширень файлів.

Статичні маршрути зазвичай визначаються як розширення файлу (наприклад, *.html), і Content-Type такі файли мають обслуговуватися.

Кілька розширень файлів включено за замовчуванням і повинні задовольняти основні потреби:

- *.html (текст/html)
- *.css (текст/css)
- *.js (текст/javascript)
- *.json (програма/json)
- *.gif (зображення/gif)
- *.jpg (зображення/jpeg)
- *.jpeg (зображення/jpeg)
- *.png (зображення/png)
- *.svg (зображення/svg+xml)
- *.ttf (шрифт/ttf)

Нативний підхід Espressif також можна використовувати, якщо ви віддаєте перевагу, але з ним важче працювати. У більшості випадків обидві схеми можуть співіснувати без проблем.

При використанні нативного підходу підтримується зіставлення символів узагальнення URI.

Можуть бути зареєстровані динамічні маршрути, які під час доступу клієнта призведуть до виклику функції Lua.

Потім ця функція може генерувати будь-який відповідний вміст, наприклад, отримувати значення датчика та повертати його.

Зауважте, що якщо ви записуєте дані датчиків у файли та обслуговуєте ці файли статично, ви будете сприйнятливі до умов змагання, коли вміст файлу може бути недоступним ззовні.

Це пов’язано з тим, що веб-сервер працює у власному потоці FreeRTOS і обслуговує файли безпосередньо з цього потоку одночасно з віртуальною машиною Lua, що працює як зазвичай.

Тому безпечніше обслуговувати такий вміст на динамічному маршруті, навіть якщо цей маршрут лише читає файл і обслуговує його.

Приклад такої установки:

```
function handler(req)
  local f = io.open('/path/to/mysensordata.csv', 'r')
  return {
    status = "200 OK",
    type = "text/plain",
    getbody = function()
      local data = f:read(512) -- pick a suiteable chunk size here
      if not data then f:close() end
      return data
    end,
  }
end
httpd.dynamic(httpd.GET, "/mysensordata", handler)
```

httpd.start()	Запускає веб-сервер.
httpd.stop()	Зупиняє веб-сервер.
httpd.static()	Реєструє статичний обробник маршруту.
httpd.dynamic()	Реєструє обробник динамічного маршруту.

httpd.unregister()	Скасовує реєстрацію попередньо зареєстрованого обробника.

ОПИС ФУНКЦІЙ API МОДУЛЯ HTTPD

httpd.start()¶

Запускає веб-сервер. Перед налаштуванням маршрутів сервер має бути запущено.

Синтаксис¶

```
httpd.start({
  webroot = "<static file prefix>",
  max_handlers = 20,
  auto_index = httpd.INDEX_NONE || httpd.INDEX_ROOT || httpd.INDEX_ALL,
})
```

Параметри¶

Надається єдина таблиця конфігурації з такими можливими полями:

- `webroot` (обов'язковий) Це встановлює префікс, який використовується під час обслуговування статичних файлів.

  Наприклад, якщо `webroot` встановлено значення "web", запит HTTP для "/index.html" призведе до того, що модуль httpd намагатиметься обслуговувати файл "web/index.html" із файлової системи.

  НЕ встановлюйте для цього значення порожній рядок, оскільки це забезпечить віддалений доступ до всієї вашої віртуальної файлової системи, включаючи спеціальні файли, такі як файли віртуальних пристроїв (наприклад, "/dev/uart1"), які, ймовірно, створять серйозну проблему безпеки.
- `max_handlers` (необов'язково) Налаштовує максимальну кількість обробників маршрутів, які підтримуватиме сервер. Значення за замовчуванням - 20, яке включає як стандартні обробники статичних розширень файлів, так і будь-які надані користувачем обробники.

Підвищення цього призведе до використання трохи додаткової пам'яті. Налаштуйте, якщо і коли це необхідно.

- auto_index Встановлює режим індексатора, який буде використовуватися. Більшість веб-серверів автоматично шукають файл "index.html", коли запитується каталог.

  Наприклад, якщо ваш веб-браузер вперше перейти на веб-сайт, наприклад http://www.example.com/, фактичний запит надійде через "/", який, у свою чергу, зазвичай перекладається як "/index". html" на сервері. Цю поведінку також можна ввімкнути в цьому модулі.

  Передбачено три режими:

  - httpd.INDEX_NONE Немає автоматичного перекладу на "index.html".
  - httpd.INDEX_ROOT Тільки корінь ("/") перекладається на "/index.html".
  - httpd.INDEX_ALL До будь-якого шляху, який закінчується на "/", додається "index.html". Наприклад, запит для «subdir/» стане «subdir/index.html», що, у свою чергу, може призвести до обслуговування файлу «web/subdir/index.html» (якщо webroot було встановлено значення «web»). Значення за замовчуванням - httpd.INDEX_ROOT .

Повернення¶

nil

приклад¶

```
httpd.start({ webroot = "web", auto_index = httpd.INDEX_ALL })
```

httpd.stop()¶

Зупиняє веб-сервер. Усі зареєстровані обробники маршрутів видалено.

Синтаксис¶

```
httpd.stop()
```

Параметри¶

Жодного.

Повернення¶

`nil`

httpd.static()¶

Реєструє статичний обробник маршруту.

Синтаксис¶

```
httpd.static(route, content_type)
```

Параметри¶

- `route` Префікс маршруту. Зазвичай у формі *.ext для статичної обробки всіх файлів із розширенням ".ext".
- `content_type` Значення, яке потрібно надіслати в `Content-Type` заголовку для цього типу файлу.

Повернення¶

Код помилки в разі невдачі або `nil` успіху. Код помилки – це значення, яке повертає функція `httpd_register_uri_handler()`.

приклад¶

```
httpd.start({ webroot = "web" })
httpd.static("*.csv", "text/csv") -- Serve CSV files under web/
```

***httpd.dynamic()*¶**

Реєструє обробник динамічного маршруту.

Синтаксис¶

```
httpd.dynamic(method, route, handler)
```

Параметри¶

- `method` Метод HTTP, до якого застосовується цей маршрут. Один з:
- `httpd.GET`
- `httpd.HEAD`
- `httpd.PUT`
- `httpd.POST`
- `httpd.DELETE`
- `route` Префікс маршруту. Пам'ятайте про будь-який кінцевий знак «/», оскільки це може взаємодіяти з функціональністю `auto_index`.
- `handler` Функція обробки маршруту - `handler(req)`. Наданий об'єкт запиту `req` має такі поля/функції:
- `method` Метод запиту. Те саме, що `method` параметр вище. Якщо одна функція зареєстрована для кількох методів, це поле можна використовувати для визначення методу, використаного запитом.
- `uri` Запитаний URI. Включає як шлях, так і рядок запиту (якщо є).
- `query` Сам по собі рядок запиту. Не розшифровано.
- `headers` Табличний об'єкт, у якому можна шукати заголовки запитів. Зауважте, що через те, що API Espressif не надає способу перебору всіх

заголовків, ця таблиця відображатиметься порожньою, якщо передати її до `pairs()`.

- `getbody()` Функція, яку можна викликати для поступового читання в тілі запиту. Розмір кожного блоку встановлюється за допомогою параметра Kconfig «Розмір блоку отримання тіла». Коли ця функція повертається, `nil` кінець тіла досягнуто. Може викликати помилку, якщо з якоїсь причини не вдається прочитати тіло (наприклад, час очікування, помилка мережі).

Зауважте, що наданий `req` об'єкт *дійсний лише* в межах цього єдиного виклику обробника. Спроби зберегти запит і використати його пізніше *будуть* невдалими.

Повернення¶

Таблиця з даними відповіді для надсилання запитуючому клієнту:

```
{
  status = "200 OK",
  type = "text/plain",
  headers = {
    ['X-Extra'] = "My custom header value"
  },
  body = "Hello, Lua!",
  getbody = dynamic_content_generator_func,
}
```

Підтримувані поля:

- `status` Код стану та рядок для надсилання. Якщо не включено, використовується "200 OK". Іншими поширеними рядками можуть бути «404 не знайдено», «400 неправильний запит» і «500 внутрішня помилка сервера».

- `type` Значення для `Content-Type` заголовка. Компонент веб-сервера Espressif обробляє цей заголовок спеціально, тому він надається тут, а не в таблиці `headers`. - `body` Повний вміст для надсилання.

- getbody Функція для джерела вмісту тіла, подібно до того, як зчитується тіло запиту. Ця функція буде викликатися неоднаразово, а повернутий рядок від кожного виклику надсилатиметься клієнту як фрагмент.

Після повернення цієї функції nil тіло вважається завершеним і подальші виклики функції не здійснюватимуться. Гарантується, що функція буде викликана, доки вона не повернеться, nil навіть якщо під час надсилання вмісту виникне помилка.

Потрібно вказати лише один із body і . getbody

приклад¶

```
httpd.start({ webroot = "web" })

function put_foo(req)
  local body_len = tonumber(req.headers['content-length']) or 0
  if body_len < 4096
  then
    local f = io.open("/upload/foo.txt", "w")
    local body = req.getbody()
    while body
    do
      f:write(body)
      body = req.getbody()
    end
    f:close()
    return { status = "201 Created" }
  else
    return { status = "400 Bad Request" }
  end
end
```

httpd.unregister()¶

Скасовує реєстрацію попередньо зареєстрованого обробника. Обробники за замовчуванням можуть бути незареєстрованими.

Синтаксис¶

```
httpd.unregister(method, route)
```

Параметри¶

- `method` Метод, для якого зареєстровано маршрут. Один з:
- `httpd.GET`
- `httpd.HEAD`
- `httpd.PUT`
- `httpd.POST`
- `httpd.DELETE`
- `route` Префікс маршруту.

Повернення¶

`1` на успіх, `nil` на невдачу (в тому числі, якщо маршрут не був зареєстрований).

приклад¶

Скасування реєстрації одного зі стандартних статичних обробників:

```
httpd.start({ webroot = "web" })
httpd.unregister(httpd.GET, "*.jpeg")
```

МОДУЛЬ COAP

Модуль CoAP забезпечує просту реалізацію відповідно до протоколу CoAP . Основна серверна частина кінцевої точки базується на microcoap і багатьох інших посиланнях на код libcoap .

Цей модуль реалізує як клієнтську, так і серверну частину. GET/PUT/POST/DELETE частково підтримується клієнтом. Сервер може реєструвати функції та змінні Lua. Спостереження чи виявлення ще не підтримуються.

Константи	Константи для різних функцій.
coap.Client()	Створює клієнт CoAP.
coap.Server()	Створює сервер CoAP.
coap.client:get()	Видає серверу запит GET.
coap.client:put()	Видає серверу запит PUT.
coap.client:post()	Видає серверу запит POST.
coap.client:delete()	Видає серверу запит DELETE.
coap.server:listen()	Запускає сервер CoAP на вказаному порту.
coap.server:close()	Закриває сервер CoAP.
coap.server:var()	Реєструє змінну Lua як кінцеву точку на сервері.
coap.server:func()	Реєструє функцію Lua як кінцеву точку на сервері.

Константи

Константи для різних функцій.

`coap.CON`, `coap.NON` представляють типи запитів.

`coap.TEXT_PLAIN`, `coap.LINKFORMAT`,

coap.XML, coap.OCTET_STREAM, представляють типи вмісту coap.EXI. coap.JSON

coap.Client()

Створює клієнт CoAP.

Синтаксис

coap.Client()

Параметри

немає

Повернення

Клієнт CoAP

приклад

```
cc = coap.Client()
-- assume there is a coap server at ip 192.168.100
cc:get(coap.CON, "coap://192.168.18.100:5683/.well-known/core")
-- GET is not complete, the result/payload only print out in console.
cc:post(coap.NON, "coap://192.168.18.100:5683/", "Hello")
```

coap.Server()

Створює сервер CoAP.

Синтаксис

coap.Server()

Параметри

немає

Повернення

Сервер CoAP

приклад

```
-- use copper addon for firefox
cs=coap.Server()
cs:listen(5683)

myvar=1
```

```
cs:var("myvar") -- get coap://192.168.18.103:5683/v1/v/myvar will return
the value of myvar: 1

all='[1,2,3]'
cs:var("all", coap.JSON) -- sets content type to json

-- function should tack one string, return one string.
function myfun(payload)
  print("myfun called")
  respond = "hello"
  return respond
end
cs:func("myfun") -- post coap://192.168.18.103:5683/v1/f/myfun will call
myfun
```

КЛІЄНТ COAP

coap.client:get()

Видає серверу запит GET.

Синтаксис

`coap.client:get(type, uri[, payload])`

Параметри

- `type` `coap.CON`, `coap.NON`, за замовчуванням CON. Якщо тип CON і запит не вдається, бібліотека повторює ще чотири рази, перш ніж відмовитися.
- `uri` URI, як-от "coap://192.168.18.103:5683/v1/v/myvar", підтримуються лише IP-адреси, тобто без розпізнавання імені хоста.
- `payload` необов'язково, корисне навантаження буде розміщено в розділі корисного навантаження запиту.

Повернення

`nil`

coap.client:put()

Видає серверу запит PUT.

Синтаксис

`coap.client:put(type, uri[, payload])`

Параметри

- `type` `coap.CON`, `coap.NON`, за замовчуванням CON. Якщо тип CON і запит не вдається, бібліотека повторює ще чотири рази, перш ніж відмовитися.
- `uri` URI, як-от "coap://192.168.18.103:5683/v1/v/myvar", підтримуються лише IP-адреси, тобто без розпізнавання імені хоста.
- `payload` необов'язково, корисне навантаження буде розміщено в розділі корисного навантаження запиту.

Повернення

`nil`

coap.client:post()

Видає серверу запит POST.

Синтаксис

`coap.client:post(type, uri[, payload])`

Параметри

- `type` coap.CON, coap.NON, за замовчуванням CON. якщо тип CON і запит не вдасться, запит буде повторено ще 4 рази, перш ніж відмовитися.
- `uri` uri, наприклад coap://192.168.18.103:5683/v1/v/myvar, підтримується лише IP.
- `payload` необов'язково, корисне навантаження буде розміщено в розділі корисного навантаження запиту.

Повернення

`nil`

coap.client:delete()

Видає серверу запит DELETE.

Синтаксис

`coap.client:delete(type, uri[, payload])`

Параметри

- `type` `coap.CON`, `coap.NON`, за замовчуванням CON. Якщо тип CON і запит не вдається, бібліотека повторює ще чотири рази, перш ніж відмовитися.
- `uri` URI, як-от "coap://192.168.18.103:5683/v1/v/myvar", підтримуються лише IP-адреси, тобто без розпізнавання імені хоста.
- `payload` необов'язково, корисне навантаження буде розміщено в розділі корисного навантаження запиту.

Повернення

`nil`

СЕРВЕР COAP

coap.server:listen()

Запускає сервер CoAP на вказаному порту.

Синтаксис

`coap.server:listen(port[, ip])`

Параметри

- `port` порт сервера (номер)
- `ip` додаткова IP-адреса

Повернення

`nil`

coap.server:close()

Закриває сервер CoAP.

Синтаксис

`coap.server:close()`

Параметри

немає

Повернення

`nil`

coap.server:var()

Реєструє змінну Lua як кінцеву точку на сервері. тоді значення змінної може бути отримано клієнтом за допомогою методу GET, представленого клієнту як URI . Шлях кінцевої точки для змінної – '/v1/v/'.

Синтаксис

`coap.server:var(name[, content_type])`

Параметри

- `name` ім'я змінної Lua
- `content_type` необов’язковий, за замовчуванням `coap.TEXT_PLAIN`, див. Узгодження вмісту

Повернення

`nil`

приклад

```
-- use copper addon for firefox
cs=coap.Server()
cs:listen(5683)

myvar=1
```

```
cs:var("myvar") -- get coap://192.168.18.103:5683/v1/v/myvar will return
the value of myvar: 1
-- cs:var(myvar), WRONG, this api accept the name string of the varialbe.
but not the variable itself.
all='[1,2,3]'
cs:var("all", coap.JSON) -- sets content type to json
```

coap.server:func()

Реєструє функцію Lua як кінцеву точку на сервері. Потім функція може бути викликана клієнтом за допомогою методу POST. представлений як URI для клієнта. Шлях кінцевої точки для функції – «/v1/f/».

Коли клієнт надсилає запит POST до цього URI, корисне навантаження буде передано функції як параметр.

Повернене функцією значення буде корисним навантаженням у повідомленні для клієнта.

Зареєстрована функція ПОВИННА приймати ЛИШЕ ОДИН параметр рядкового типу та повертати ОДНЕ значення рядка або не повертати нічого.

Синтаксис

`coap.server:func(name[, content_type])`

Параметри

- `name` ім'я функції Lua
- `content_type` необов'язковий, за замовчуванням `coap.TEXT_PLAIN`, див. Узгодження вмісту

Повернення

`nil`

приклад

```
-- use copper addon for firefox
cs=coap.Server()
cs:listen(5683)

-- function should take only one string, return one string.
function myfun(payload)
  print("myfun called")
  respond = "hello"
  return respond
```

```
end
cs:func("myfun") -- post coap://192.168.18.103:5683/v1/f/myfun will call
myfun
-- cs:func(myfun), WRONG, this api accept the name string of the function.
but not the function itself.
```

МОДУЛЬ TLS

Модуль TLS залежить від модуля мережі , це обов'язкова залежність.

NodeMCU містить версію бібліотеки mbed TLS з відкритим кодом .

З конфігурацією NodeMCU за замовчуванням він підтримує TLS 1.2 із підтримкою більшості поширених функцій. Зокрема, він забезпечує:

- шифри: AES, Camellia
- режими ланцюжка: CBC, CFB, CTR, GCM
- алгоритми дайджесту: RIPEMD-160, SHA1, SHA2
- алгоритми підпису: RSA, детермінований ECDSA
- алгоритми обміну ключами: DHE і ECDHE
- еліптичні криві: secp{256,384}r1, secp256k1, bp{256,384}.

УВАГА

Суворі обмеження пам'яті ESP8266 означають, що tls модуль набагато краще підходить для зв'язку з користувальницькими, спеціально створеними кінцевими точками з невеликими ланцюжками сертифікатів ніж із Інтернетом загалом.

За замовчуванням наша конфігурація mbedTLS запитує фрагменти TLS розміром не більше 4 КБ і не бажає обробляти фрагментовані повідомлення, тобто весь ServerHello, який включає ланцюжок сертифікатів сервера, має відповідати цьому обмеженню.

Квіткування TLS дуже інтенсивно використовує купу, вимагаючи від 25 до 30 **кілобайт**, навіть із нашими зменшеними розмірами буфера.

Цей модуль обробляє перевірку сертифіката, коли використовується SSL/TLS.

tls.createConnection()	Створює з'єднання TLS.
tls.socket:close()	Закриває розетку.
tls.socket:connect()	Підключіться до віддаленого сервера.
tls.socket:getpeer()	Отримати порт та ip однорангового пристрою.

tls.socket:hold()	Гальмуйте прийом даних шляхом розміщення запиту на блокування функції отримання TCP.
tls.socket:on()	Зареєструйте функції зворотного виклику для певних подій.
tls.socket:send()	Надсилає дані віддаленому вузлу.
tls.socket:unhold()	Розблокувати отримання даних TCP шляхом скасування попереднього hold().
tls.cert.verify()	Керує процесом перевірки сертифіката, коли NodeMCU встановлює безпечне з'єднання.
tls.cert.auth()	Контролює ключ і сертифікат клієнта, які використовуються, коли ESP створює з'єднання TLS (наприклад, через tls.

ОПИС ФУНКЦІЙ API МОДУЛЯ TLS

tls.createConnection()

Створює з'єднання TLS.

Синтаксис

```
tls.createConnection()
```

Параметри

немає

Повернення

підмодуль tls.socket

приклад

```
tls.createConnection()
```

МОДУЛЬ TLS.SOCKET

tls.socket:close()

Закриває розетку.

Синтаксис

`close()`

Параметри

немає

Повернення

`nil`

Дивись також

`tls.createConnection()`

tls.socket:connect()

Підключіться до віддаленого сервера.

Синтаксис

`connect(port, ip|domain)`

Параметри

- `port` номер порту
- `ip` IP-адреса або рядок доменного імені

Повернення

`nil`

Дивись також

`tls.socket:on()`

tls.socket:getpeer()

Отримати порт та ip однорангового пристрою.

Синтаксис

`getpeer()`

Параметри

немає

Повернення

- `ip` однолітка
- `port` однолітка

tls.socket:hold()

Гальмуйте прийом даних шляхом розміщення запиту на блокування функції отримання TCP. Цей запит не діє негайно, Espressif рекомендує викликати його, резервуючи 5*1460 байт пам'яті.

Синтаксис

`hold()`

Параметри

немає

Повернення

`nil`

Дивись також

`tls.socket:unhold()`

tls.socket:on()

Зареєструйте функції зворотного виклику для певних подій.

Синтаксис

on(event, function())

Параметри

- event рядок, який може бути "dns", "connection", "reconnection", "disconnection", "receive" або "sent"
- function(tls.socket[, string]) функція зворотного виклику. Перший параметр - розетка. Якщо подією є "receive", другим параметром є отримані дані у вигляді рядка. Якщо подією є "повторне підключення", другий параметр є причиною помилки підключення (рядок). Якщо подією є "dns", другий параметр буде або nil або рядковим відтворенням розв'язаної адреси.

Повернення

nil

приклад

```
srv = tls.createConnection()
srv:on("receive", function(sck, c) print(c) end)
srv:on("connection", function(sck, c)
  -- Wait for connection before sending.
  sck:send("GET / HTTP/1.1\r\nHost: google.com\r\nConnection: keep-
alive\r\nAccept: */*\r\n\r\n")
end)
srv:connect(443,"google.com")
```

Дивись також

- tls.createConnection()
- tls.socket:hold()

tls.socket:send()

Надсилає дані віддаленому вузлу.

Синтаксис

`send(string)`

Параметри

- `string` дані в рядку, які будуть надіслані на сервер

Повернення

`nil`

Примітка

Кілька послідовних `send()` викликів не гарантовано працюватимуть (і часто не працюють), оскільки мережеві запити розглядаються SDK як окремі завдання. Натомість підпишіться на подію "sent" у сокеті та надішліть додаткові дані (або закрийте) у цьому зворотному виклику.

Дивись також

`tls.socket:on()`

tls.socket:unhold()

Розблокувати отримання даних TCP шляхом відкликання попереднього `hold()`.

Синтаксис

`unhold()`

Параметри

немає

Повернення

`nil`

Дивись також

`tls.socket:hold()`

МОДУЛЬ TLS.CERT

tls.cert.verify()

Керує процесом перевірки сертифіката, коли NodeMCU встановлює безпечне з’єднання.

Синтаксис

`tls.cert.verify(enable)`

`tls.cert.verify(pemdata[, pemdata])`

`tls.cert.verify(callback)`

Параметри

- `enable` Логічне значення, яке вказує, чи слід увімкнути перевірку чи ні. За умовчанням під час завантаження є `false`.
- `pemdata` Рядок, що містить сертифікат ЦС для перевірки. Таких може бути декілька.
- `callback` Функція Lua, яка повертає ключі TLS і сертифікати для використання з підключеннями. Зворотний виклик повинен очікувати один цілочисельний аргумент; для значення k зворотний виклик повинен повернути k-й сертифікат ЦС (у формі DER або PEM), який він бажає використати для перевірки віддаленої кінцевої точки, або `nil` якщо такого сертифіката ЦС не існує. Якщо сертифікати не повернуто, пристрій не перевірить віддалену кінцеву точку.

Повернення

`true` якщо це спрацювало.

Може викликати ряд помилок, якщо подано недійсні дані.

приклад

Встановіть безпечне https-з’єднання та переконайтеся, що ланцюжок сертифікатів дійсний.

```
tls.cert.verify(true)
http.get("https://example.com/info", nil, function (code, resp)
print(code, resp) end)
```

Завантажте сертифікат у флеш-чіп і зробіть запит. Це сертифікат IdenTrust DST Root CA X3 ; він використовується, наприклад, letsencrypt для їх проміжних повноважень X3; letsencrypt - це один із варіантів безкоштовного отримання власних сертифікатів SSL.

```
tls.cert.verify([[
-----BEGIN CERTIFICATE-----
MIIDSjCCAjKgAwIBAgIQRK+wgNajJ7qJMDmGLvhAazANBgkqhkiG9w0BAQUFADA/
MSQwIgYDVQQKExtEaWdpdGFsIFNpZ25hdHVyZSBUcnVzdCBDby4xFzAVBgNVBAMT
DkRTVCBSb290IENBIFgzMB4XDTAwMDkzMDIxMTIxOVoXDTIxMDkzMDE0MDExNVow
PzEkMCIGA1UEChMbRGlnaXRhbCBTaWduYXR1cmUgVHJ1c3QgQ28uMRcwFQYDVQQD
Ew5EU1QgUm9vdCBDQSBYMzCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEB
AN+v6ZdQCINXtMxiZfaQguzH0yxrMMpb7NnDfcdAwRgUi+DoM3ZJKuM/IUmTrE4O
rz5Iy2Xu/NMhD2XSKtkyj4zl93ewEnu1lcCJo6m67XMuegwGMoOifooUMM0RoOEq
OLl5CjH9UL2AZd+3UWODyOKIYepLYYHsUmu5ouJLGiifSKOeDNoJjj4XLh7dIN9b
xiqKqy69cK3FCxolkHRyxXtqqzTWMIn/5WgTe1QLyNau7Fqckh49ZLOMxt+/yUFw
7BZy1SbsOFU5Q9D8/RhcQPGX69Wam40dutolucbY38EVAjqr2m7xPi71XAicPNaD
aeQQmxkqtilX4+U9m5/wAl0CAwEAAaNCMEAwDwYDVR0TAQH/BAUwAwEB/zAOBgNV
HQ8BAf8EBAMCAQYwHQYDVR0OBBYEFMSnsaR7LHH62+FLkHX/xBVghYkQMA0GCSqG
SIb3DQEBBQUAA4IBAQCjGiybFwBcqR7uKGY3Or+Dxz9LwwmglSBd49lZRNI+DT69
ikugdB/OEIKcdBodfpga3csTS7MgROSR6cz8faXbauX+5v3gTt23ADq1cEmv8uXr
AvHRAosZy5Q6XkjEGB5YGV8eAlrwDPGxrancWYaLbumR9YbK+rlmM6pZW87ipxZz
R8srzJmwN0jP41ZL9c8PDHIyh8bwRLtTcm1D9SZImlJnt1ir/md2cXjbDaJWFBM5
JDGFoqgCWjBH4d1QB7wCCZAA62RjYJsWvIjJEubSfZGL+T0yjWW06XyxV3bqxbYo
Ob8VZRzI9neWagqNdwvYkQsEjgfbKbYK7p2CNTUQ
-----END CERTIFICATE-----
]])

http.get("https://letsencrypt.org/", nil, function (code, resp)
print(code, resp) end)
```

Примітки

Сертифікат, необхідний для перевірки, зберігається у флеш-чіпі. Виклик tls.cert.verify з true дозволяє перевірити значення, збережене у флеш-пам’яті.

Сертифікат можна завантажити в мікросхему флеш-пам’яті двома способами: один під час створення мікропрограми, а інший – під час початкового завантаження мікропрограми.

Щоб завантажити сертифікат під час збірки, просто розмістіть файл із сертифікатом ЦС (у форматі PEM) у `server-ca.crt` кореневій папці дерева збірки мікропрограми nodemcu.

Сценарії збірки включать це в кінцевий образ мікропрограми.

Альтернативний підхід є легшим для розробки, а саме надання даних PEM у вигляді рядкового значення до `tls.cert.verify`.

Це збереже сертифікат у флеш-пам’яті та ввімкне перевірку для цього сертифіката.

Наступні завантаження ESP можуть використовувати `tls.cert.verify(true)` та використовувати збережений сертифікат.

Версія `callback` на основі -перевизначає інформацію у флеш-пам’яті, доки не буде скасовано реєстрацію зворотного виклику *або* не буде створено одну з інших форм виклику.

tls.cert.auth()

Контролює ключ і сертифікат клієнта, які використовуються, коли ESP створює з'єднання TLS (наприклад, через `tls.createConnection` або `https` або `MQTT` з'єднання з `secure = true`).

Синтаксис

`tls.cert.auth(enable)`

`tls.cert.auth(pemdata[, pemdata])`

`tls.cert.auth(callback)`

Параметри

- `enable` Логічне значення, яке вказує, чи будуть наступні підключення TLS надавати сертифікат клієнта. За умовчанням під час завантаження є `false`.
- `pemdata` Два рядки, перший із яких містить сертифікат клієнта в PEM-кодуванні, а другий - закритий ключ клієнта в PEM-кодуванні.
- `callback` Функція Lua, яка повертає ключі TLS і сертифікати для використання з підключеннями. Зворотний виклик повинен очікувати один цілочисельний аргумент; якщо це 0, зворотній виклик має повернути закритий ключ пристрою. В іншому випадку для аргументу k зворотний виклик має повернути k-й сертифікат (у формі DER або PEM) у ланцюжку сертифікатів пристроїв.

Повернення

`true` якщо це спрацювало.

Може викликати ряд помилок, якщо подано недійсні дані.

приклад

Відкрийте клієнт MQTT.

```
tls.cert.auth(true)
tls.cert.verify(true)

m = mqtt.Client('basicPubSub', 1500, "admin", "admin", 1)
```

Завантажте сертифікат у флеш-чіп.

```
tls.cert.auth([[
-----BEGIN CERTIFICATE-----
CLIENT CERTIFICATE String (PEM file)
-----END CERTIFICATE-----
]]
,
[[
-----BEGIN RSA PRIVATE KEY-----
CLIENT PRIVATE KEY String (PEM file)
-----END RSA PRIVATE KEY-----
]])
```

Примітки

Сертифікат, необхідний для перевірки, зберігається у флеш-чіпі. Виклик `tls.cert.auth` з `true` дає змогу перевірити значення, збережене у флеш-пам’яті.

Сертифікат не можна визначити під час створення мікропрограми, але його можна завантажити у флеш-пам’ять під час початкового завантаження мікропрограми.

Його можна надати, передавши дані PEM як рядкове значення до `tls.cert.auth`. Сертифікат буде збережено у флеш-чіпі та ввімкнено перевірку з цим сертифікатом.

Наступні завантаження ESP можуть використовувати `tls.cert.auth(true)` та використовувати збережений сертифікат.

Версія callback на основі -перевизначає інформацію у флеш-пам'яті, доки не буде скасовано реєстрацію зворотного виклику *або* не буде створено одну з інших форм виклику.

Функція tls.setDebug

mbedTLS можна скомпілювати з підтримкою налагодження. Якщо так, функція tls.setDebug відображається на mbedtls_debug_set_threshold функції та може використовуватися для ввімкнення або вимкнення викиду налагодження на консоль. Додаткову інформацію дивіться в документації mbedTLS.

МОДУЛЬ WEBSOCKET

Клієнтський модуль websocket , який реалізує RFC6455 (версія 13) і забезпечує простий інтерфейс для надсилання та отримання повідомлень.

Реалізація підтримує фрагментовані повідомлення, автоматично відповідає на запити ping і періодично ping, якщо сервер не зв'язується.

Підтримка SSL/TLS

Зверніть увагу на обмеження, задокументовані в модулі net .

websocket.createClient()	Створює новий клієнт websocket.
websocket.client:close()	Закриває підключення через веб-сокет.
websocket.client:config(params)	Налаштовує екземпляр клієнта websocket.
websocket.client:connect()	Намагається встановити підключення через веб-сокет до вказаної URL-адреси.
websocket.client:on()	Реєструє функцію зворотного виклику для обробки подій websockets (для кожного типу події може бути зареєстрована лише одна функція обробки).
websocket.client:send()	Надсилає повідомлення через веб-сокет.

ОПИС ФУНКЦІЙ API МОДУЛЯ WEBSOCKET

websocket.createClient()

Створює новий клієнт websocket. Цей клієнт має зберігатися у змінній і надаватиме всі функції для обробки з'єднання.

Коли з'єднання закривається, той самий клієнт усе ще можна використовувати повторно – функції зворотного виклику зберігаються – і ви можете знову підключитися до будь-якого сервера.

Перед утилізацією клієнта обов'язково зателефонуйте `ws:close()`.

Синтаксис

`websocket.createClient()`

Параметри

немає

Повернення

`websocketclient`

приклад

```
local ws = websocket.createClient()
-- ...
ws:close()
ws = nil
```

websocket.client:close()

Закриває підключення через веб-сокет. Клієнт створює закритий фрейм і намагається елегантно закрити веб-сокет. Якщо сервер не відповідає, з'єднання припиняється після невеликого часу очікування.

Цю функцію можна викликати, навіть якщо веб-сокет не підключено.

Цю функцію *завжди* потрібно викликати перед видаленням посилання на клієнт websocket.

Синтаксис

`websocket:close()`

Параметри

немає

Повернення

`nil`

приклад

```
ws = websocket.createClient()
ws:close()
ws:close() -- nothing will happen

ws = nil -- fully dispose the client as Lua will now gc it
```

websocket.client:config(params)

Налаштовує екземпляр клієнта websocket.

Синтаксис

`websocket:config(params)`

Параметри

- `params` таблиця з параметрами конфігурації. Розпізнаються такі ключі:
- `headers` таблиця додаткових заголовків запитів, що впливають на кожен запит

Повернення

`nil`

приклад

```
ws = websocket.createClient()
ws:config({headers={['User-Agent']='NodeMCU'}})
```

websocket.client:connect()

Намагається встановити підключення через веб-сокет до вказаної URL-адреси.

Синтаксис

`websocket:connect(url)`

Параметри

- `url` URL для websocket.

Повернення

`nil`

приклад

```
ws = websocket.createClient()
ws:connect('ws://echo.websocket.org')
```

Якщо це не вдається, повідомлення про помилку буде доставлено через `websocket:on("close", handler)`.

websocket.client:on()

Реєструє функцію зворотного виклику для обробки подій websockets (для кожного типу події може бути зареєстрована лише одна функція обробки).

Синтаксис

`websocket:on(eventName, function(ws, ...))`

Параметри

- `eventName` тип події websocket для реєстрації функції зворотного виклику. Ці події: `connection`, `receive` і `close`.
- `function(ws, ...)` функція зворотного виклику. Першим параметром функції завжди є websocketclient. Інші аргументи потрібні залежно від типу події. Додаткову інформацію див. у прикладі. Якщо `nil`, будь-який попередньо налаштований зворотний виклик буде скасовано.

Повернення

`nil`

приклад

```
local ws = websocket.createClient()
ws:on("connection", function(ws)
  print('got ws connection')
end)
ws:on("receive", function(_, msg, opcode)
  print('got message:', msg, opcode) -- opcode is 1 for text message, 2
for binary
end)
ws:on("close", function(_, status)
  print('connection closed', status)
  ws = nil -- required to Lua gc the websocket client
end)
```

Зауважте, що зворотній виклик закриття також запускається, якщо виникає будь-яка помилка.

Код стану для закриття, якщо не 0, це означає помилку, як описано в наступній таблиці.

Код статусу	Пояснення
0	Користувач подав запит на закриття або з’єднання було розірвано
-1	Не вдалося отримати протокол із URL-адреси
-2	Ім'я хосту завелике (>256 символів)
-3	Недійсний номер порту (має бути >0 і <= 65535)
-4	Не вдалося отримати ім’я хоста
-5	DNS не вдалося знайти ім’я хоста
-6	Сервер запросив припинення
-7	Сервер надіслав недійсну відповідь HTTP рукостискання (тобто сервер надіслав неправильний ключ)

Код статусу	Пояснення
від -8 до -14	Не вдалося виділити пам'ять для отримання повідомлення
-15	Сервер не відповідає бітовому протоколу FIN належним чином
-16	Не вдалося виділити пам’ять для надсилання повідомлення
-17	Сервер не перемикає протоколи
-18	Час очікування підключення
-19	Сервер не відповідає на перевірки працездатності та не спілкується
від -99 до -999	error

websocket.client:send()

Надсилає повідомлення через веб-сокет.

Синтаксис

`websocket:send(message, opcode)`

Параметри

- `message` дані для надсилання.
- `opcode` за бажанням установіть код операції (за замовчуванням: 1, текстове повідомлення)

Повернення

`nil` або помилка, якщо розетка не підключена

приклад

```
ws = websocket.createClient()
ws:on("connection", function()
  ws:send('hello!')
end)
```

МОДУЛЬ GGOSSIP

Цей модуль базується на протоколі переговорів і може використовуватися для розповсюдження інформації через мережу на інші вузли.

Час, необхідний для того, щоб інформація досягла всіх вузлів, становить logN. Для кожного круглого числа n інформацію отримають 2^n вузлів.

Синтаксис

```
gossip = require('gossip')
```

Звільнення

```
gossip.inboundSocket:close()
```

```
gossip = nil
```

Використання

```
config = {
    seedList = { '192.168.0.1', '192.168.0.15' },
    debug = true,
    debugOutput = print
}
gossip = require ("gossip")
gossip.setConfig(config)
gossip.start()
```

Стратегія

Кожен контролер випадковим чином вибере IP зі свого початкового списку. Він надішле `SYN` запит на цю IP-адресу та встановить вузол-одержувач `state` у проміжний стан між `Up` та `Suspect`.

Вузол, який отримує `SYN` запит, обчислить різницю між отриманим networkState і власним networkState. Потім він надішле цю різницю як `ACK` запит.

Якщо немає даних для надсилання, буде надіслано лише ACK. Коли ACK буде отримано, стан відправника повернеться до стану Up, а приймаючий вузол оновить свій власний стан мережі за допомогою diff (на основі ACK відповіді).

Gossip встановлює, чи містить інформація, отримана з іншого вузла, більш свіжі дані, спочатку порівнюючи revision, потім heartbeat і нарешті state. Стани, які є ближчими до того, щоб DOWN мати пріоритет як автономний вузол, не оновлюють його пульс.

Будь-який інший параметр можна надіслати разом із обов'язковим revision, heartbeat і state таким чином користувач зможе поширювати інформацію по мережі.

Кожного разу, коли вузол отримує «свіжі» дані, gossip.updateCallback буде викликано з цими даними як першим параметром.

Наразі не реалізовано видалення вузлів, які не працюють, за винятком того факту, що їхній статус позначається як REMOVE.

Приклад використання

У мережі є кілька модулів, які вимірюють температуру. Ми хочемо знати максимальну та мінімальну температуру в певний час і щоб кожен вузол відображав їх.

Рішення грубої сили полягало б у тому, щоб зробити запит до кожного вузла з однієї точки та зберегти значення min та max, а потім повернутися до кожного вузла та надати їм обчислені min та max.

Для цього потрібно n*2 раундів, де n - кількість вузлів. Він також відкриває алгоритм до однієї точки відмови (вузол, який відповідає за збір даних).

Використовуючи gossip, можна змусити вузол надсилати своє останнє значення через SYN або pushGossip() використовувати callbackUpdate функцію для порівняння значень інших вузлів зі своїми власними.

Виходячи з цього, вузол відображатиме значення, про які він знає, пліткуючи з іншими. Дані будуть передаватися в ~log(n) раундах, де n – кількість вузлів.

Терміни

revision: генерація вузла; якщо вузол перезапускається, версія буде збільшена на одиницю. Дані версії зберігаються у вигляді файлу для забезпечення сталості

heartBeat: час роботи вузла в секундах (tmr.time()). Це використовується, щоб допомогти іншим вузлам визначити, чи є інформація про цей конкретний вузол новішою.

networkState: список із станом мережі, що складається з ip ключа та revision, heartBeat а state також значень, упакованих у таблицю.

state: усі вузли починаються зі стану, встановленого на UP і коли вузол надсилає SYN запит, він позначатиме вузол призначення у проміжному стані, доки не отримає від нього ACK або а SYN. Якщо вузол отримує будь-яке повідомлення, він позначає IP-адресу відправника, оскільки UP це забезпечує підтвердження того, що вузол онлайн.

setConfig()

Синтаксис

```
gossip.setConfig(config)
```

Встановлює конфігурацію для пліток. Доступні варіанти:

`seedList`: список насіння пліток почнеться з; це буде оновлено, коли будуть виявлені нові вузли. Достатньо, щоб усі вузли починалися з однакового IP-адреси в seedList, оскільки як тільки вони отримають одне спільне початкове значення, дані поширюватимуться.

Якщо seedList порожній, надсилається широкомовне повідомлення, тож це можна використовувати для автоматичного виявлення вузлів.

`roundInterval`: інтервал у мілісекундах, за який gossip вибере випадковий вузол із початкового списку та надішле `SYN` запит

`comPort`: порт для прослуховуючого UDP-сокета

`debug`: прапорець, який надаватиме повідомлення про налагодження

`debugOutput`: якщо для параметра debug встановлено значення `true`, тоді цей метод використовуватиметься як зворотний виклик із повідомленням про налагодження як першим параметром

```
config = {
    seedList = {'192.168.0.54','192.168.0.55'},
    roundInterval = 10000,
    comPort = 5000,
    debug = true,
    debugOutput = function(message) print('Gossip says: '..message); end
}
```

Якщо жодне з них не вказано, значення за замовчуванням:

`seedList`: нуль

`roundInterval`: 10000 (10 секунд)

`comPort`: 5000

debug : помилковий

debugOutput : друк

start()

Синтаксис

```
gossip.start()
```

callbackFunction

Синтаксис

```
gossip.callbackFunction = function(data)
  processData(data)
end

-- stop the callback
gossip.callbackFunction = nil
```

Якщо оголошено, ця функція буде викликатися кожного разу, коли є SYN з новими даними.

pushGossip()

Синтаксис

```
gossip.pushGossip(data, [ip])

-- remove data
gossip.pushGossip(nil, [ip])
```

setRevManually()

Синтаксис

```
gossip.setRevFileValue(number)
```

Єдиний сценарій, коли rev потрібно встановлювати вручну, це коли до мережі додається новий вузол із такою ж IP-адресою.

Наявність меншої версії, ніж у попереднього вузла з таким самим IP-адресою, змусить пліткарів вважати отримані дані старими, таким чином ігноруючи їх.

getNetworkState()

Синтаксис

```
networkState = gossip.getNetworkState()
print(networkState)
```

Повернення

Рядок у форматі JSON щодо стану мережі.

приклад:

```
{
  "192.168.0.53": {
    "state": 3,
    "revision": 25,
    "heartbeat": 2500,
    "extra" : "this is some extra info from node 53"
  },
  "192.168.0.75": {
    "state": 0,
    "revision": 4,
    "heartbeat": 6500
  }
}
```

МОДУЛЬ ОНОВЛЕННЯ OTA

Модуль OTA Upgrade надає доступ до підтримки IDF Over-The-Air Upgrade, що дозволяє застосовувати та завантажувати нове мікропрограмне забезпечення програми.

Цей модуль не стосується того, звідки походить нова програма. Вибір джерела та методу завантаження (наприклад, https, tftp) залишається за користувачем, як і тригер для запуску оновлення.

Загальний підхід полягає в тому, щоб пристрій періодично перевіряв центральний сервер і порівнював наданий номер версії з поточною версією, а за необхідності запускав оновлення.

Щоб використовувати `otaupgrade` модуль, має існувати принаймні два розділи OTA (тип `app`, підтип `ota_0` / `ota_1`), а також розділ «otadata» (тип `data`, підтип `ota`).

IDF реалізує типовий «тригерний» підхід до оновлень, коли один із розділів містить запущену програму, а оновлення завантажується в неактивний розділ і лише після повного завантаження та перевірки воно позначається як завантажувальне.

Це робить систему стійкою до неповних оновлень, будь то через втрату живлення, переривання завантажень або інші подібні речі.

Приклад таблиці розділів для OTA може виглядати так:

```
# Name,   Type, SubType, Offset,  Size
nvs,      data, nvs,     0x9000,  0x5000
otadata,  data, ota,     0xe000,  0x2000
ota_0,    app,  ota_0,   0x10000, 0x130000
ota_1,    app,  ota_1,  0x140000, 0x130000
```

Залежно від того, чи був встановлений завантажувач зібраний із підтримкою відкату чи без неї, сам процес оновлення складається з чотирьох або трьох кроків.

Без підтримки відкату кроки такі:

- `otaupgrade.commence()`
- `otaupgrade.write(data)` розділити нове зображення програми
- `otaupgrade.complete(1)` завершити та перезавантажити нову програму

Якщо завантажувач створено з підтримкою відкату, після завантаження нової програми (і перевірки на «хорошу» за будь-якими показниками, які вибере користувач) потрібен додатковий крок:

- `otaupgrade.accept()` щоб позначити цей образ як дійсний і дозволити його завантаження знову.

Якщо нову мікропрограму не буде `accept()` оновлено до перезавантаження пристрою, завантажувач повернеться до попередньої версії мікропрограми (за умови, що цей завантажувач побудовано з підтримкою відкату).

Загальний тест перед позначенням нової мікропрограми як дійсної полягає в тому, щоб переконатися, що сервер оновлення доступний, виходячи з того, що якщо мікропрограму можна оновити віддалено, вона «достатньо хороша», щоб прийняти її.

otaupgrade.info()	За допомогою цієї функції можна отримати інформацію про завантаження, стан і версію програми.
otaupgrade.commence()	Очищає резервний розділ програми та готується отримати нову мікропрограму програми.
otaupgrade.write(дані)	Запишіть фрагмент даних програмного забезпечення до правильного розділу та місця.
otaupgrade.complete (перезавантаження)	Завершує оновлення та, за бажанням, негайно перезавантажує нову мікропрограму програми.

otaupgrade.accept()	Якщо встановлений завантажувач створено з підтримкою відкату, новий образ програми за замовчуванням завантажується лише один раз.
otaupgrade.rollback()	Нове мікропрограмне забезпечення може вирішити, що воно не працює належним чином, і запросити явний відкат до попередньої версії.

***otaupgrade.info()*¶**

За допомогою цієї функції можна отримати інформацію про завантаження, стан і версію програми. Зазвичай він використовується для перевірки версії запущеної програми, порівняння з «бажаною» версією, щоб вирішити, чи потрібне оновлення.

Синтаксис¶

`otaupgrade.info()`

Параметри¶

Жодного.

Повернення¶

Список із трьох значень:

- ім'я розділу запущеної програми
- ім'я розділу, позначеного для наступного завантаження (зазвичай те саме, що й запущена програма, але після `otaupgrade.complete()` неї може вказувати на новий розділ програми.
- таблиця, ключі якої є іменами OTA-розділів, а відповідні значення – це таблиці, що містять:
  - `state` один із `new`, `testing`, `valid`,

    або `invalid`, `aborted` можливо `undefined`, .

    Цінності `invalid` та `aborted` здебільшого означають те саме.

    Перегляньте документацію IDF, щоб дізнатися більше. Розділ

у testing стані має викликати, otaupgrade.accept() якщо він хоче стати valid.

- name назва програми, зазвичай "NodeMCU"
- date дата побудови
- time час побудови
- version версія збірки, встановлена змінною *PROJECT_VER* під час збірки
- secure_version номер безпечної версії, якщо безпечне завантаження ввімкнено
- idf_version версія IDF

приклад¶

```
boot_part, next_part, info = otaupgrade.info()
print("Booted: "..boot_part)
print("  Next: "..next_part)
for p,t in pairs(info) do
  print("@ "..p..":")
  for k,v in pairs(t) do
    print("    "..k..": "..v)
  end
end
print("Running version: "..info[boot_part].version)
```

otaupgrade.commence()¶

Очищає резервний розділ програми та готується отримати нову мікропрограму програми.

Якщо підтримку відкату ввімкнено, зауважте, що запущену програму потрібно спочатку позначити як дійсну/прийняту, перш ніж можна буде розпочати нове оновлення OTA.

Синтаксис¶

otaupgrade.commence()

Параметри¶

Жодного.

Повернення¶

`nil`

Помилка Lua може виникнути, якщо через певну причину не вдається розпочати оновлення OTA (наприклад, через неправильне налаштування розділу).

***otaupgrade.write(дані)*¶**

Запишіть фрагмент даних програмного забезпечення до правильного розділу та місця. Дані повинні передаватись послідовно, IDF не підтримує дані, що не відповідають порядку, як це було б, наприклад, у випадку з bittorrent.

Синтаксис¶

```
otaupgrade.write(data)
```

Параметри¶

- `data` рядок двійкових даних

Повернення¶

`nil`

Помилка Lua може виникнути, якщо дані не можуть бути записані, наприклад через те, що дані не є дійсним OTA-образом (IDF виконує деякі перевірки щодо цього).

***otaupgrade.complete (перезавантаження)*¶**

Завершує оновлення та, за бажанням, негайно перезавантажує нову мікропрограму програми.

Синтаксис¶

```
otaupgrade.complete(reboot)
```

Параметри¶

- reboot 1, щоб негайно перезавантажити нову мікропрограму, нуль, щоб продовжити роботу

Повернення¶

nil

Помилка Lua може виникнути, якщо зображення не пройшло перевірку або в зображення взагалі не було записаних даних.

приклад¶

```
-- Quick, dirty and totally insecure "push upgrade" for development use.
-- Use netcat to push a new firmware to a device:
--   nc -q 1 your-device-ip 9999 < build/NodeMCU.bin
--
osv = net.createServer()
osv:listen(9999, function(conn)
  print('Commencing OTA upgrade')
  local status, err = pcall(otaupgrade.commence)
  if err then
    print(err)
    conn:send(err)
    conn:close()
  end
  conn:on('receive', function(sck, data)
    status, err = pcall(function() otaupgrade.write(data) end)
    if err then
      print(err)
      conn:send(err)
      conn:close()
    end
  end)
  conn:on('disconnection', function()
    print('EOF, completing OTA')
    status, err = pcall(function() otaupgrade.complete(1) end)
    if err then
      print(err)
    end
  end)
end)
```

***otaupgrade.accept()*¶**

Якщо встановлений завантажувач створено з підтримкою відкату, новий образ програми за замовчуванням завантажується лише один раз.

Під час цього «тестового запуску» він може виконувати будь-які відповідні перевірки (наприклад, перевіряти, чи може він все ще досягти сервера оновлення), і, якщо він задоволений, може позначити себе як дійсний.

Якщо система не буде позначена як дійсна, під час наступного перезавантаження система буде «відкочуватися» до попередньої версії.

Синтаксис¶

```
otaupgrade.accept()
```

Параметри¶

Жодного.

Повернення¶

```
nil
```

***otaupgrade.rollback()*¶**

Нове мікропрограмне забезпечення може вирішити, що воно не працює належним чином, і запросити явний відкат до попередньої версії.

Якщо виклик цієї функції вдасться, система перезавантажиться без повернення з виклику.

Зверніть увагу, що також можна повернутися до попередньої версії мікропрограми навіть після виклику нової версії `otaupgrade.accept()`.

Синтаксис¶

```
otaupgrade.rollback()
```

Параметри¶

Жодного.

4. БЕЗДРОТОВА МЕРЕЖА "EASY NET EVERYWHERE"

Призначення

Бездротова мережа "Easy Net Everywhere" призначена для застосування в територіально-розподілених системах, які потребують наявності бездротового зв'язку з об'єктами управління з гарантованою доставкою команд і даних зі швидкістю 250 Kbps у радіусі до 10 км (залежно від умов навколишньої місцевості та складу обладнання).

Область застосування

Мережа "Easy Net Everywhere" може бути використана для вирішення завдань у проектах різного ступеня складності, на Приклад:

- для управління як окремими об'єктами, так і роєм об'єктів, на Приклад, дронами, роботами, роботизованими платформами та іншими механізмами;
- для використання в якості фреймворку для побудови різних систем і реалізації проектів рівня "Internet Of Things";
- для отримання і пересилання в мережу координат GPS, створення GPS трекерів і маячків тощо;
- для використання в системах зовнішньої та внутрішньої охорони об'єктів і приміщень, управління камерами відеоспостереження, у системах тривожної сигналізації, у метеостанціях і системах моніторингу датчиків показників навколишнього середовища;
- для віддаленого керування станом об'єктів у комунальних службах, на Приклад, водопровідними засувками, кранами, клапанами тощо;
- для збирання та передавання інформації з віддалених автономних датчиків, датчиків пристроїв автоматики, медичних приладів тощо;
- для систем керування дозаторами, зерносушарками, теплицями тощо;
- для систем автоматичного керування мікрокліматом, поливом, освітленням тощо;

- для систем керування міськими світлофорами з метою підвищення комфорту пересування автомобільного транспорту, зниження рівня ДТП тощо;
- для систем управління класу "Smart Home», ввімкнення/вимкнення побутових приладів і електроприводів, для управління потужністю в навантаженні тощо;
- для швидкої реалізації територіально-розподілених DIY-проектів (Arduino тощо) ;
- для створення текстових месенджерів і чатів.

Область застосування мережі обмежується тільки її технічними характеристиками.

Технічні характеристики:

Частотний діапазон: ISM.

Радіоканали: 0-125 (від 2400 до 2525 МГц).

Полоса пропускання радіоканалу - 1МГц.

Трансивери: NRF24L01+, NRF24L01+PA+LNA, E01-ML01DP5, E01-2G4M27D.

Відстань приймання/передавання даних у межах прямої видимості на висоті 1 м на швидкості 250 Kbps:

- трансивер NRF24L01 - 100 м;
- трансивер NRF24L01+PA+LNA, E01-ML01DP5 - 1км;
- трансивер E01-2G4M27D - 2 км.

Швидкість передачі даних: 250 Kbps, 1 Mbps і 2 Mbps.

Максимальна довжина пакета: 256 байт.

Час передачі пакета 256 байт на швидкості 250 Kbps - 50 мс (5 Кб/с).

Інтерфейси користувача:

- UART 9600 - 921600 baud;
- Bluetooth: BLE.

Інтерфейси контролера мережі:

- I^2C;
- SPI;
- GPS UART 115200 baud.

Кількість функціональних виводів контролера – 14.

Функції виводів контролера (призначаються користувачем):

- ADC - 14;
- DAC - 2;
- PWM - 11;
- Servo - 11;
- GPIO: вихід - 11, вхід - 14, input only - 3;
- GPIO переривання – 14.

Архітектури мережі:

Архітектура "багато-до-багатьох":

- кількість вузлів – 254.

Кластерна архітектура:

- кластерів - 14;
- кількість вузлів у кластері - 15;
- кількість роутерів - 30.

Напруга живлення: 3,3 В, 5 В.

Струм споживання вузла мережі:

- в активному режимі: 40 ма;
- у пасивному режимі: 25 мка.

Діапазон робочих температур: -40°C...+85°C.

Особливості мережі

Мережа "Easy Net Everywhere" має кластерну архітектуру з маршрутизацією пакетів і можливістю масштабування.

Вузлами мережі використовуються трансивери невеликої потужності для забезпечення обміну даними тільки там, де це необхідно, не займаючи ефір там, де це небажано.

Таке рішення забезпечує створення оптимальної топології мережі, підвищення порога виявлення роботи трансиверів і зменшення ймовірності виникнення колізій. Розв'язання колізій у мережі реалізується пропрієтарним протоколом.

Вузли мережі можуть одночасно працювати як у режимі точка-точка на невеликій дистанції в межах прямої видимості до 2 км, так і в режимі ретрансляції пакетів, що дає можливість встановлювати зв'язок між об'єктами на відстані до 10 км, а через мережу StarLink, мережу GSM і GSM-Internet - у встановлених межах.

Передавання даних здійснюється каналами, які обираються випадковим чином із числа доступних. Теоретично доступно 125 каналів з шириною смуги пропускання 1 МГц. Практично - залежить від конкретних умов.

Усі вузли мережі рівнозначні і можуть мати повний або обмежений доступ до інших вузлів мережі залежно від використаної архітектури.

Кожен вузол мережі може передати команди і дані будь-якому іншому вузлу.

Адміністратор мережі може встановити необхідні обмеження для кожного вузла. Адміністрування мережі можливе з будь-якого вузла, наділеного повноваженнями.

Кожна наступна транзакція не використовує повторно канал попередньої транзакції, що ускладнює виявлення вузла сканерами ефіру. Тривалість однієї транзакції на одному каналі не перевищує 25 мс.

Девіація каналів передавання даних може бути встановлена адміністратором у межах +/- 2...60 від частоти основного каналу.

Адміністратор може встановити обмежений список робочих каналів, при цьому кожному адресату може бути поставлено у відповідність один або кілька фіксованих/змінних каналів.

Безпека. Кожен вузол мережі має адресу мережі, адресу вузла, пароль і логін мережі, а також пароль доступу до вузла через Bluetooth.

Шифрування даних: пропрієтарний алгоритм маскування даних.

Передача даних у мережі може ініціюватися:

- користувачем;
- вузлом мережі;
- подією;
- перевищенням/зниженням значення параметра відносно заданого порога;
- після закінчення часу таймера одноразово або із заданою періодичністю.

Для передавання даних вузлу-одержувачу використовується маршрутизація відправника. Для кожного вузла-одержувача можна вказати свій маршрут проходження пакетів і записати його в профіль адресата.

Така маршрутизація детермінована і для стаціонарних і малорухомих об'єктів оптимальніша, ніж стохастична маршрутизація в mesh-мережі.

Взаємодія користувача з будь-яким вузлом мережі здійснюється за допомогою послідовного інтерфейсу UART на швидкості до 921600 baud або за допомогою інтерфейсу Bluetooth (BLE).

Конфігурація та встановлення параметрів вузлів мережі здійснюється за допомогою AT- команд через бездротовий інтерфейс Bluetooth (BLE) планшета або смартфона та дротовий інтерфейс UART.

Усі налаштування зберігаються в пам'яті вузла.

Порівняльні характеристики малопотужних бездротових мереж представлені в табл.1

Обмеження:

- обмеження 1 зумовлене частотою роботи трансивера - 2.4 GHz. Зв'язок можливий у межах прямої видимості;

- обмеження 2: за надто великої дистанції між роутерами зв'язок на даній ділянці може перерватися і мережа розпадеться на кілька самостійних сегментів.

Для виключення такої ситуації кількість роутерів на одиницю площі/обсягу ареалу функціонування мережі має бути більшою за мінімально необхідну кількість. У цьому разі існує більше варіантів вибору альтернативних маршрутів проходження пакетів.

Таблиця 1 - Порівняльні характеристики малопотужних бездротових мереж

Параметр	Z-Wave	Zigbee	LoRaWAN	Easy Net Everywhere
Потужність передавача	0 dBm/1 мВт	0 dBm/1 мВт	до 30 dBm/1000 мВт	до 27 dBm/500 мВт
Дальність зв'язку між вузлами мережі	до 100 м.	до 100 м.	до 10 км (модулі класу С) з додатковим обладнанням	до 10 км (модулі класу С) • без додаткового обладнання
Швидкість передачі	до 100 Кбіт/с *	до 250 Кбіт/с *	до 19,2 Кбіт/с *	• 250 Кбіт/с ; 1 та 2 Mbit/c
Затримка передачі	Tx: 1% Rx: 99%	< 5 мс	від 1 мс до десятків секунд	• < 1 мс
Кількість каналів	16	16	127 (діапазон 2.4 GHz)	• 127
Мобільність	низька	низька	низька	• висока
Складність установки	середня	середня	висока	• низька
Орієнтація	пристрої і датчики	пристрої і датчики	пристрої і датчики	• людина, пристрої і датчики
Обмін текстовими повідомленнями	немає	немає	немає	• є, 256 байт/транзакція
Комунікаційні інтерфейси користувача	немає	немає	немає	• SPI, I2C, UART 921600 baud
Бездротові інтерфейси користувача	немає	немає	немає	• Wi-Fi, Bluetooth/BLE, GSM
Навігація	немає	немає	немає	• GPS/GLONASS/Galileo/Beidou
Управління дронами	непридатна	непридатна	непридатна	• придатна
Ліцензія	не потрібна	потрібна	не потрібна	не потрібна
Вартість	висока	середня	висока	• низька

Примітки: * - залежить від відстані між пристроями; • - перевага

2. ОПИС КОМПОНЕНТІВ І ВУЗЛІВ МЕРЕЖІ

Трансивери

У всіх вузлах мережі передбачено використання трансиверів різної потужності залежно від розв'язуваних завдань. Трансивер вузла вибирається користувачем. Трансивер або впаюється в плату вузла (рекомендується), або вставляється в роз'єм (для тестування).

Вузли мережі працюють із такими трансиверами:

- трансивери NRF24L01+, NRF24L01+PA+LNA (рис.1);
- трансивери E01-ML01DP5, E01-2G4M27D (рис.2).

Рисунок 1 - Трансивери NRF24L01+ та NRF24L01+PA+LNA

Рисунок 2 – Трансивери E01-ML01DP5 та E01-2G4M27D

Антени

З трансиверами можуть використовуватися різні антени, як штатні, так і спеціальні, з конектором SMA. Від вибору антени залежить відстань, на якій трансивери можуть забезпечити надійний зв'язок.

Штатна антена має коефіцієнт посилення 3 dBi і забезпечує впевнений зв'язок на відстані 1-2 км на відкритій місцевості на висоті 1,5 м з трансивером E01-2G4M27D (рис. 3).

Рисунок 3 - Штатна антена трансивера

Антена Yagi 10 dBi забезпечує впевнений зв'язок на відстані до 5 км на відкритій місцевості на висоті 1,5 м з трансивером E01-2G4M27D (рис. 4).

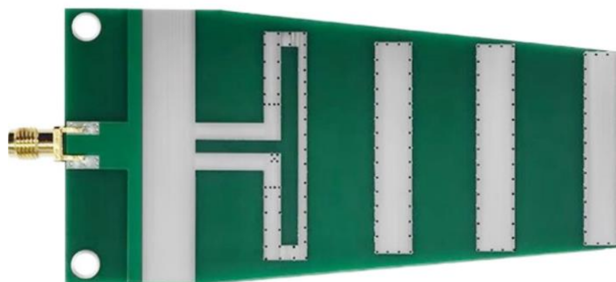


Рисунок 4 - Антена Yagi 10 dBi

Антена Yagi 25 dBi забезпечує впевнений зв'язок на відстані до 30 км на відкритій місцевості на висоті 1,5 м з трансивером E01-2G4M27D (рис. 5).

Рисунок 5 - Антена Yagi 25 dBi

Під час вибору антени слід враховувати діаграму спрямованості, яка вказується в технічних характеристиках конкретних антен.

Керування трансиверами здійснюється мікроконтролером ESP32 фірми Espressif, у керуючій програмі якого реалізовано стек системних, мережевих і користувацьких протоколів. Сукупність контролера і трансивера утворює вузол мережі.

Вибір оптимальних каналів для роботи мережі

Діапазон частот, у якому працює мережа, зумовлює певні вимоги до антен, що використовуються і вибору оптимального каналу (групи каналів) для роботи вузлів мережі.

Антени повинна мати мінімальне значення коефіцієнта стоячої хвилі КСВ (*SWR standing wave ratio*) в обраному діапазоні частот. Цей параметр визначає ККД антени і безпосередньо впливає на дальність зв'язку. Хвильовий опір антен для трансиверів становить 50 Ом.

Для визначення КСВ антен використовуються спеціальні прилади: КСВ-метри. На Приклад, у польових умовах для вимірювання необхідних параметрів антен можна використовувати LiteVNA (Potrable Vector Network Analizer).

Результати вимірювання КСВ штатної антени трансивера (рис. 3) за допомогою приладу LiteVNA представлені на рис. 6.

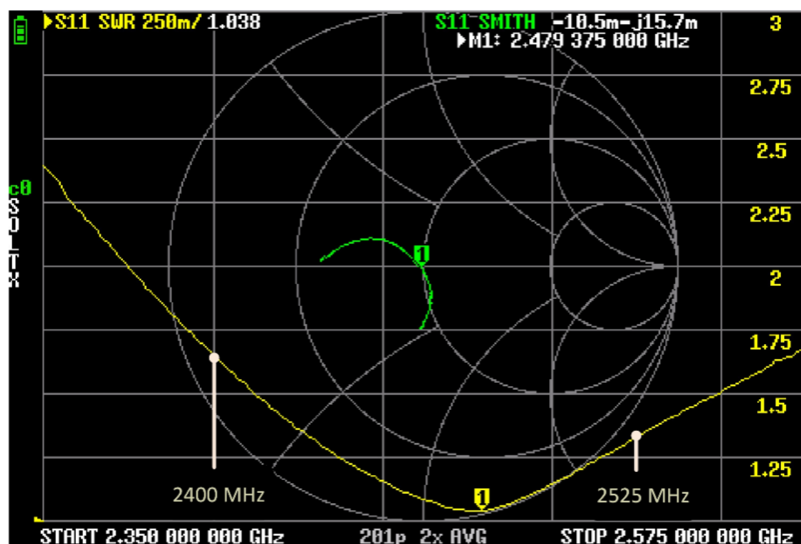


Рисунок 6 - Результати вимірювання КСВ штатної антени трансивера

З графіка випливає, що антена не є широкосмуговою. Мінімальний КСВ антени 1.038 відповідає частоті 2479 MHz або каналу з номером 79.

Прийнятний КСВ відповідає частотам 2450 MHz - 2500 MHz. Це означає, що основний канал для роботи мережі бажано вибирати в діапазоні від 50 до 100.

На рисунку 7 представлено скріншот екрана аналізатора TSA ULTRA (Tiny Spectrum Analyzer) з результатами сканування частот у діапазоні роботи мережі 2400 MHz - 2525 MHz (канали 0 - 125) під час виконання тесту для двох вузлів мережі.

На скріншоті відображено частоти, на яких здійснюється обмін даними між бездротовими пристроями:

- у діапазоні частот 2400 MHz - 2400 MHz працюють пристрої Wi-Fi;
- у діапазоні частот 2400 MHz - 2400 MHz працюють вузли мережі, займаючи канали 58 – 68;
- частота 2485 MHz відповідає основному каналу з номером 85.

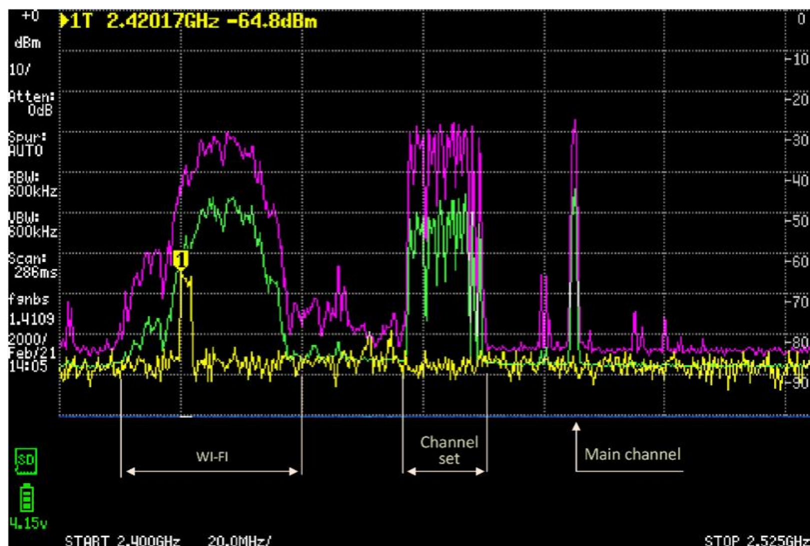


Рисунок 7 - Результат сканування діапазона 2400 MHz - 2525 MHz

Таким чином, знаючи КСВ антени, можна вибрати найбільш оптимальний діапазон частот (каналів) для роботи мережі.

У разі відсутності приладу для вимірювання КСВ антени, рекомендується обирати канали в середині робочого діапазону ISM і надалі коригувати їх на підставі статистичної інформації про помилки передавання даних за всіма обраними каналами (AT- команда AT+ERROR.log).

3. ПРИЗНАЧЕННЯ ТА СТРУКТУРА ВУЗЛІВ МЕРЕЖІ

Вузли мережі

Для побудови та організації роботи мережі використовуються мережеві функціональні вузли:

- "Net Master" (NM);
- "Controller" (C);
- "Light Node" (L);
- "Cluster Admin" (CA);
- "Net Router" (NR);
- "Repeater" (R).

Net Master

Вузол "Net Master" є головним об'єктом мережі та призначений для адміністрування мережі, надсилання команд і даних у мережу та отримання даних із мережі. Взаємодіє з мережею Internet за допомогою Wi-Fi роутера, за допомогою роутера Starlink і GSM-модема, що має підключення до UART.

Структурну схему вузла "Net Master" наведено на рис. 8.

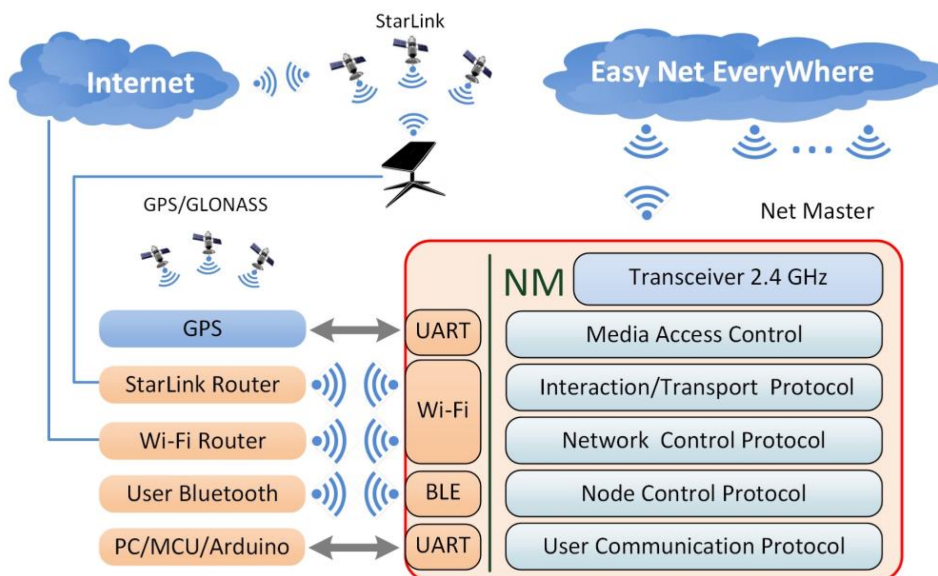


Рисунок 8 - Структурна схема вузла "Net Master"

Controller

Вузол "Controller" призначений:

- для приймання з мережі та передавання в UART команд і даних користувача, включно з текстовими повідомленнями довжиною до 256 байт;
- для виконання команд користувача;
- для збирання та передавання інформації.

Команди користувача являють собою числові значення, яким поставлено у відповідність виконання однієї дії на стороні контролера або послідовності дій, на Приклад:

- увімкнути/вимкнути будь-який пристрій;
- збільшити/зменшити потужність у навантаженні;
- повернути вал одного або декількох сервоприводів на заданий кут з заданою швидкістю;
- встановити на виводах DAC рівень постійної напруги;
- прочитати поточне значення напруги на виводах ADC і передати їх користувачеві;
- встановити тайм-аут таймера для періодичного передавання користувачеві станів виводів і значень ADC тощо.

Вузол "Controller" має можливість підключення модуля GPS. Це дає змогу в реальному масштабі часу визначати координати кожного вузла мережі в просторі, передавати координати вузлу "Net Master" з подальшим відображенням їх на Google-карті (візуалізація топології мережі на смартфоні/планшеті або комп'ютері).

Це дуже корисна можливість для завдання оптимального маршруту проходження пакетів, відображення координат об'єктів, встановлення маячків, GPS-GSM трекерів, навігації дронів, роботизованих платформ тощо.

Структурна схема вузла "Controller" представлена на рис. 9.

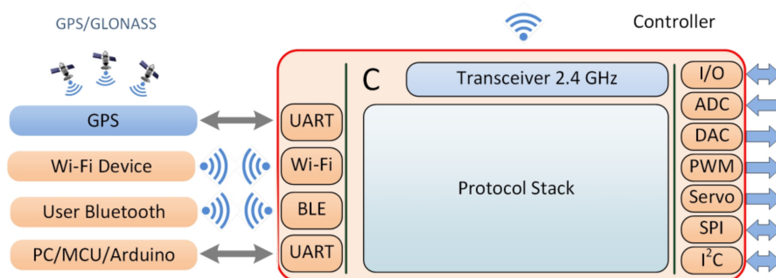


Рисунок 9 - Структурна схема вузла "Controller"

На структурній схемі вузла " Controller" з лівого боку представлені комунікаційні інтерфейси UART і Bluetooth для взаємодії користувача з контролером мережі.

З правого боку представлені послідовні системні інтерфейси SPI, I2C, і модулі контролера для управління обладнанням і отримання зовнішніх сигналів.

Light Node

Вузол "Light Node" призначений для двостороннього обміну командами і даними між інтерфейсом UART і мережею зі швидкістю до 921600 baud. Вузол не містить функціональних виводів. Для взаємодії з користувачем вузол "Light Node" має інтерфейси UART, Wi-Fi і Bluetooth (BLE).

Структурну схему вузла "Light Node" представлено на рис.10.

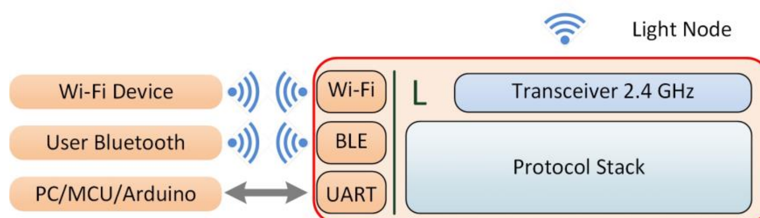


Рисунок 10 - Структурна схема вузла "Light Node"

Cluster Admin

Вузол "Cluster Admin" призначений для двостороннього обміну командами і даними користувача між інтерфейсом UART і мережею зі швидкістю до 921600 baud. Вузол здійснює адміністрування всіх вузлів кластера, виконує системні та сервісні функції для забезпечення роботи кластера. Для взаємодії з користувачем вузол "Cluster Admin" має інтерфейси UART і Bluetooth (BLE).

Структурну схему вузла "Cluster Admin" наведено на рис. 11.

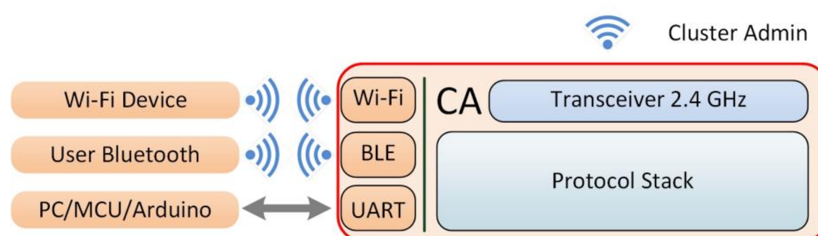


Рисунок 11 - Структурна схема вузла "Cluster Admin"

Net Router

Вузол "Net Router" призначений для маршрутизації пакетів у мережі відповідно до встановленого маршруту. Звільняє інші вузли мережі від участі в процесі маршрутизації та ретрансляції пакетів.

Реалізація вузла "Net Router" має два варіанти: одиночний і здвоєний. Одиночний роутер спочатку приймає дейтаграму, а потім передає. Оскільки режим роботи роутера симплексний, то під час передачі даних приймання даних не здійснюється. Загальний час прийому-передачі дейтаграми становить 50 мс.

У здвоєному роутері /“Fast Router”/ один роутер приймає дейтаграму, потім по інтерфейсу UART пересилає її другому роутеру, який передає дейтаграму в мережу. Час прийому-передачі дейтаграми становить 27 мс.

Для взаємодії з користувачем вузол "Net Router" має інтерфейси UART і Bluetooth (BLE).

Структурну схему одиночного вузла "Net Router" представлено на рис. 12.

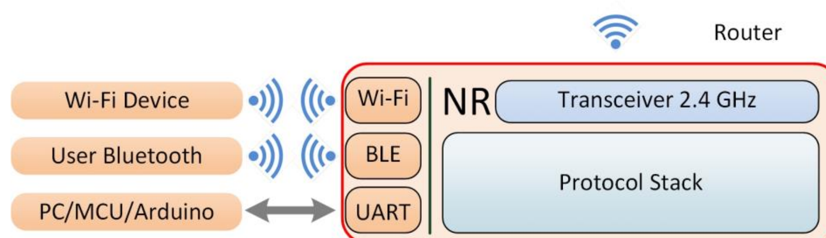


Рисунок 12 - Структурна схема одиночного вузла "Net Router"

Здвоєний роутер має дві мережеві адреси та по два інтерфейси UART і Bluetooth (BLE).

Структурна схема здвоєного вузла "Net Router" подана на рис. 13.

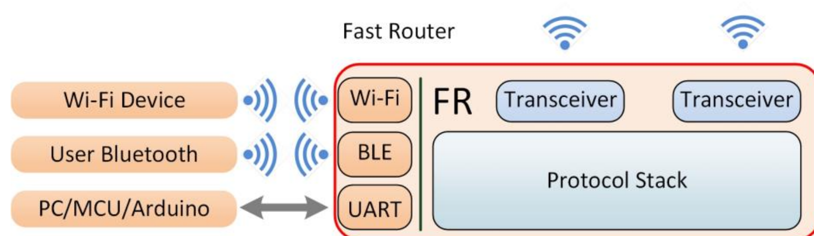


Рисунок 13 - Структурна схема здвоєного вузла "Net Router"

Repeater

Вузол "Repeater" (ретранслятор) призначений для локального розширення зони покриття мережі. Він є повільним пристроєм зі швидкістю передачі 2400 baud. Діапазон частот роботи репітера - 144 і 433 MHz.

Вузол "Repeater" доцільно використовувати в умовах поганого проходження сигналу 2,4 GHz або у відсутності можливості використання вузла "Net Router".

Вузол "Repeater" приймає і передає дейтаграму цілком по інтерфейсу UART під управлінням спеціального протоколу.

Структурну схему вузла "Repeater" наведено на рис. 14.

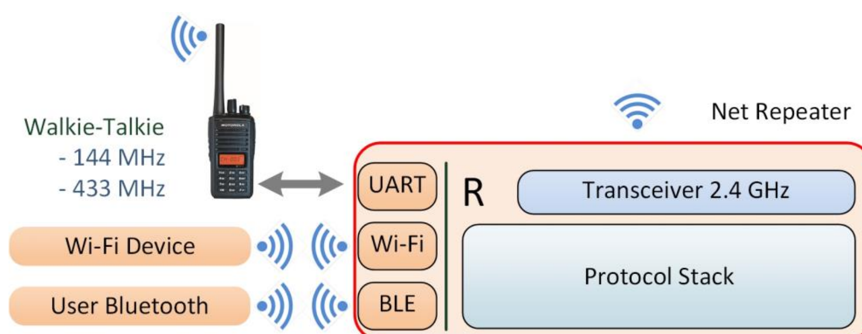


Рисунок 14 - Структурна схема вузла "Repeater"

4. АРХІТЕКТУРА, МАСШТАБУВАННЯ, АДРЕСАЦІЯ ТА РОБОТА МЕРЕЖІ

Архітектура мережі "Easy Net Everywhere" дає змогу користувачеві легко будувати необхідну топологію, оптимально відповідну для вирішення конкретного завдання.

У загальному вигляді архітектура мережі "Easy Net Everywhere" представлена на рис. 15.

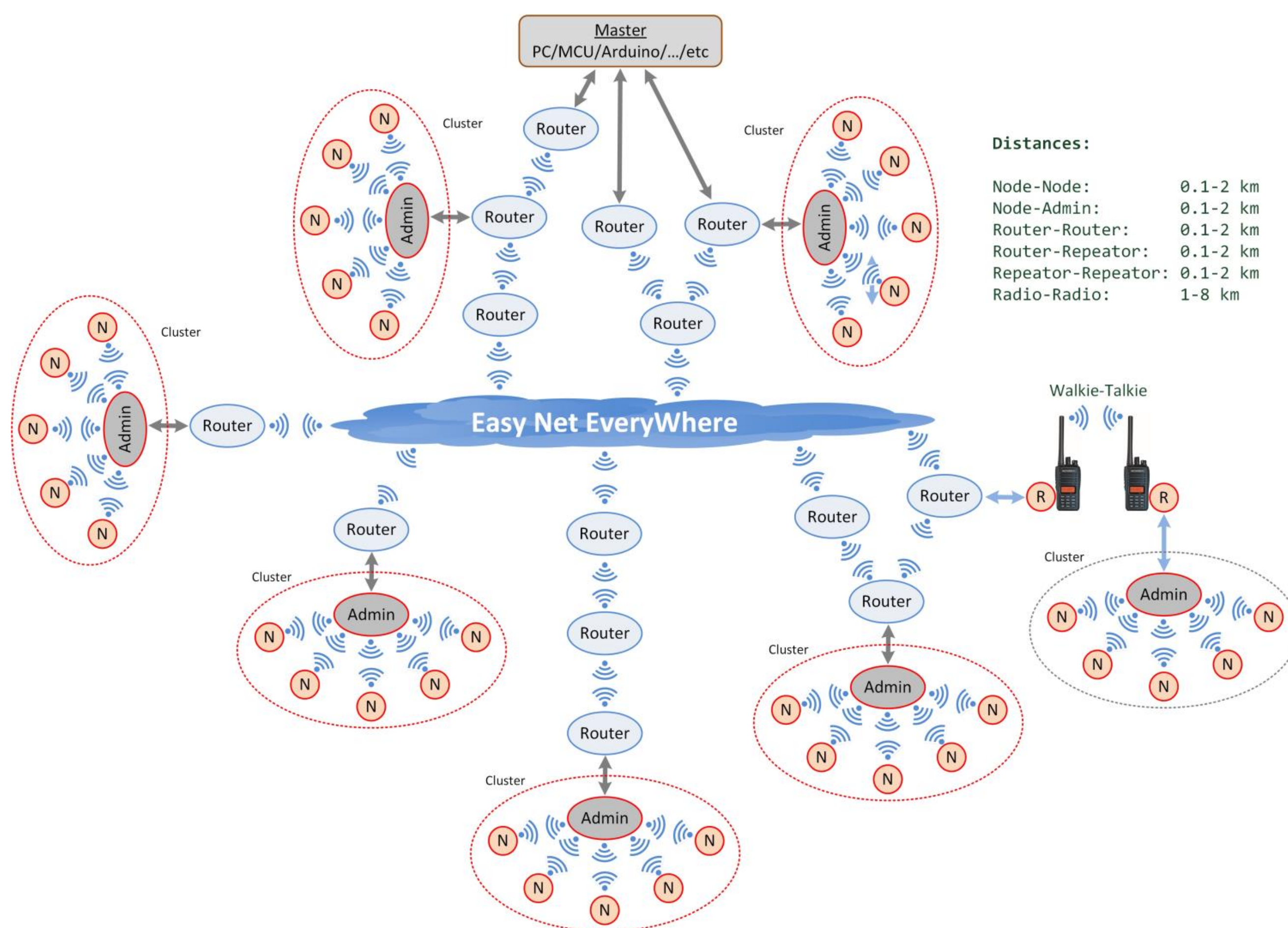


Рисунок 15 - Архітектура мережі "Easy Net Everywhere"

Масштабування мережі

Архітектура мережі "Easy Net Everywhere" є легко масштабованою.

На Приклад, для побудови системи "Розумний дім", що містить невелику кількість вузлів на невеликій площі за умов низького трафіку, застосування роутерів і репітерів недоцільне.

Однак, у територіально розподіленій системі з безліччю вузлів і відстанями між вузлами до 10 км, роутери та репітери є необхідністю.

Масштабування мережі реалізується відповідно до таких моделей:

- Модель "Simple Net";
- Модель "Light Cluster Net";
- Модель "Medium Cluster Net";
- Модель "Union Cluster Net".

Модель мережі "Simple Net" встановлює режим роботи простої мережі з аморфною архітектурою та аморфною топологією без використання роутерів та інших вузлів. У цьому режимі користувач і кожен вузол має прямий доступ до всіх інших вузлів мережі. Усі вузли за замовчуванням рівнозначні. За замовчуванням, Майстром мережі може стати будь-який вузол, до якого в цей момент підключився користувач. Тому мережа може мати кілька Майстрів мережі, якщо це не заборонено адміністратором.

Така модель рекомендується для створення дуже малих локальних мереж з невеликим трафіком.

Архітектура мережі на основі моделі "Simple Net" представлена на рис. 16.

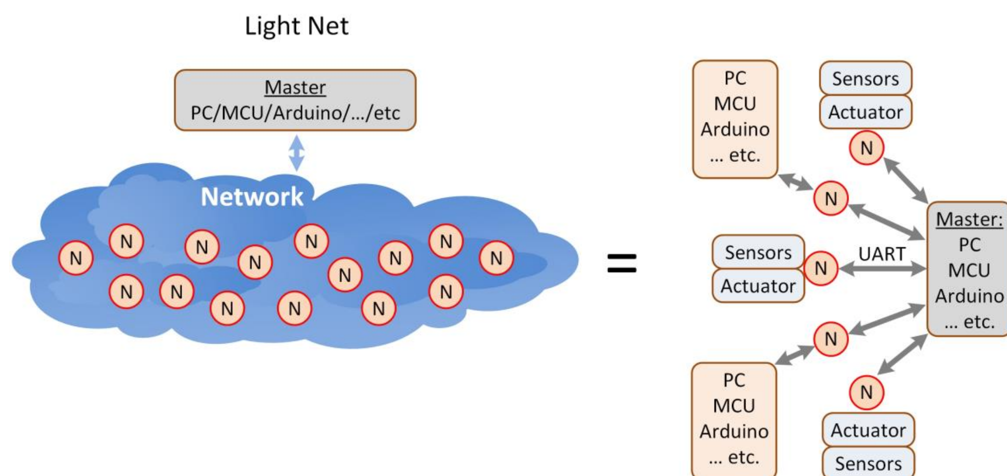


Рисунок 16 - Архітектура мережі на основі моделі "Simple Net"

Фізична реалізація мережі "Simple Net" представлена на рис. 17.

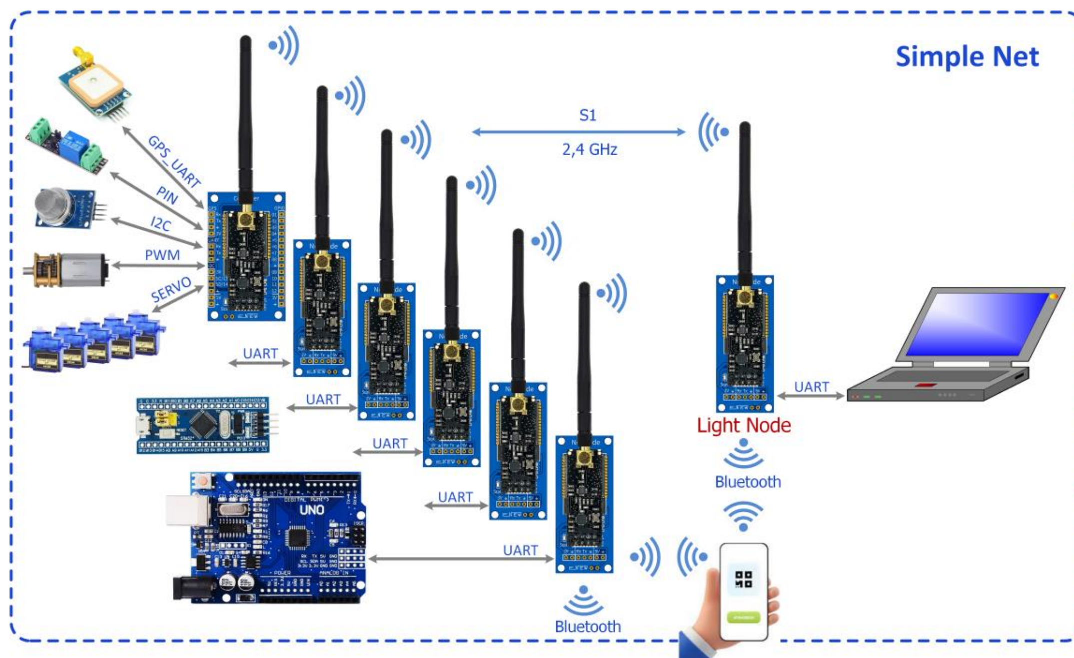


Рисунок 17 - Фізична реалізація мережі "Simple Net"

Значення дистанції S1 залежить від використовуваного трансивера та антени. На Приклад, у разі використання штатної антени і трансивера NRF24L01+PA+LNA дистанція S1 становить 1 км, а в разі використання трансивера E01-2G4M27D - 2 км.

Модель мережі "Light Cluster Net" встановлює режим роботи мережі з одним або кількома кластерами без використання роутерів. У цьому режимі доступ до вузлів кластера здійснюється через вузол "Cluster Admin", який інкапсулює вузли кластера і звільняє користувача від роботи з управління вузлами "Light Node".

Вузли "Cluster Admin" мають прямий зв'язок із вузлом "Net Master" за інтерфейсом UART зі швидкістю до 921600 baud.

Така модель рекомендується для створення невеликих локальних мереж з невеликим трафіком.

Архітектура мережі на основі моделі "Light Cluster Net" представлена на рис. 18.

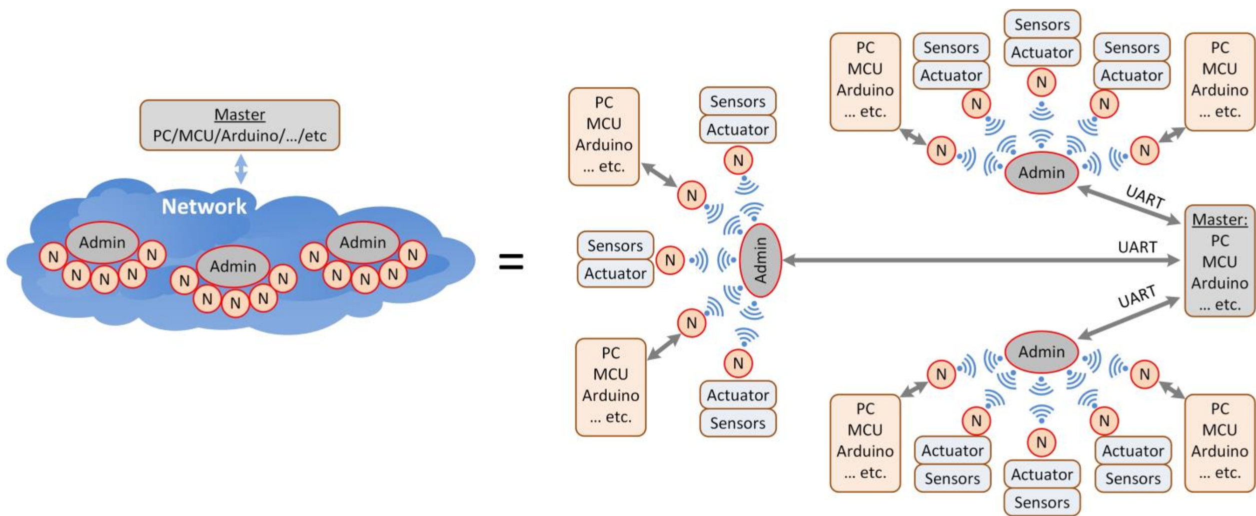


Рисунок 18 - Архітектура мережі на основі моделі "Light Cluster Net"

Фізична реалізація кластера мережі "Light Cluster Net" представлена на рис. 19.

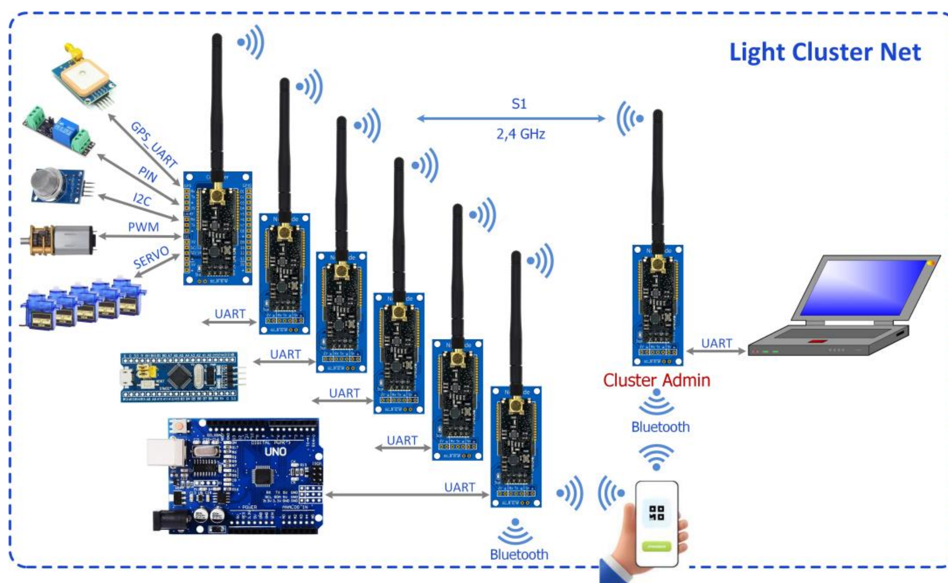


Рисунок 19 - Фізична реалізація кластера мережі "Light Cluster Net"

Модель "Medium Cluster Net" встановлює режим роботи мережі з декількома кластерами з використанням роутерів і репітерів та відносно високим трафіком в умовах наявності перешкод для проходження мережевих пакетів.

Доступ до вузлів мережі також здійснюється через вузол "Cluster Admin".

Така модель рекомендується для створення територіально-розподілених мереж з відносно високим трафіком. Фізична реалізація кластера мережі "Medium Cluster Net" представлена на рис. 20.

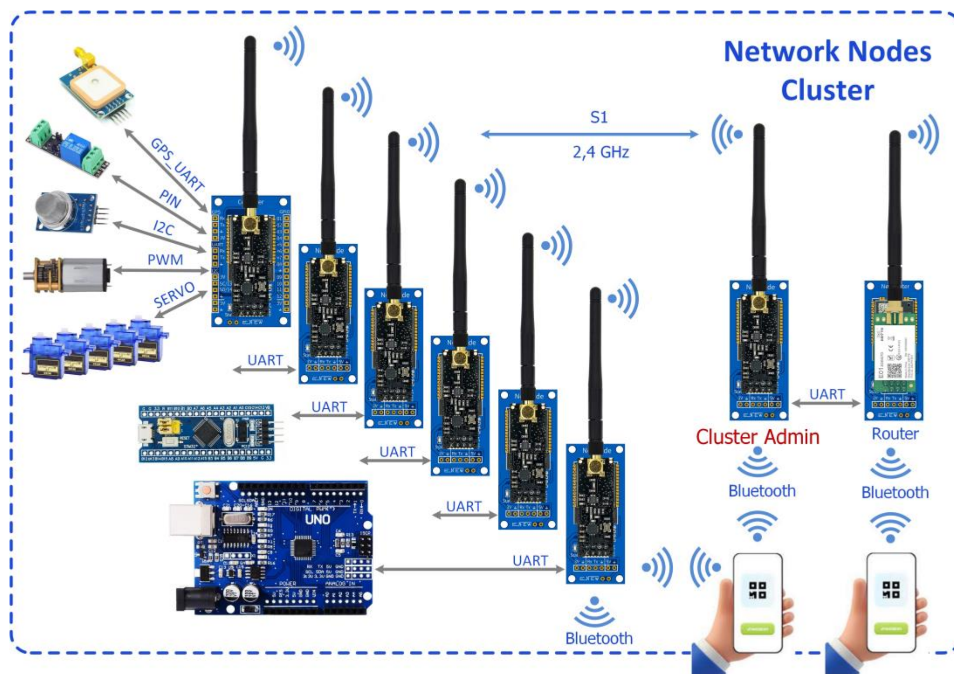


Рисунок 20 - Фізична реалізація кластера мережі "Medium Cluster Net"

Архітектура мережі на основі моделі "Medium Cluster Net" представлена на рис. 21.

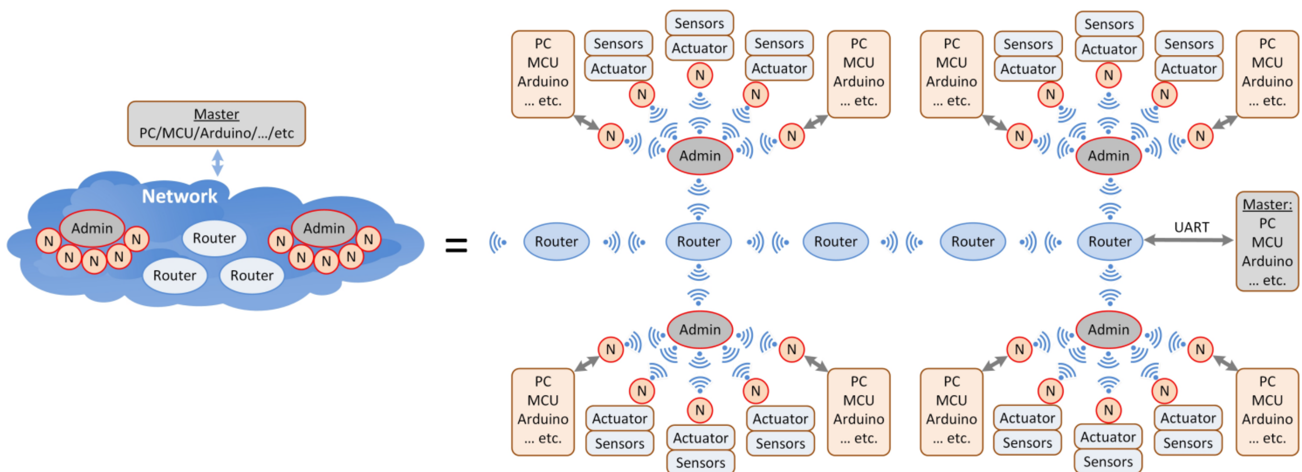


Рисунок 21 - Архітектура мережі на основі моделі "Medium Cluster Net"

Використання роутерів дає змогу користувачеві призначати та змінювати маршрути проходження пакетів, оптимально розподілити мережевий трафік і обійти об'єкти, що перешкоджають обміну даними між об'єктами мережі.

Фізична реалізація мережі "Medium Cluster Net" представлена на рис. 22.

Під час побудови мережі слід враховувати, що в разі спільного використання різних трансиверів дистанція стійкого зв'язку S2 дорівнюватиме найменшому значенню S1,

де: S1 - дистанція для трансивера NRF24L01+PA+LNA (1 км);

S2 - дистанція для трансивера E01-2G4M27D (2 км).

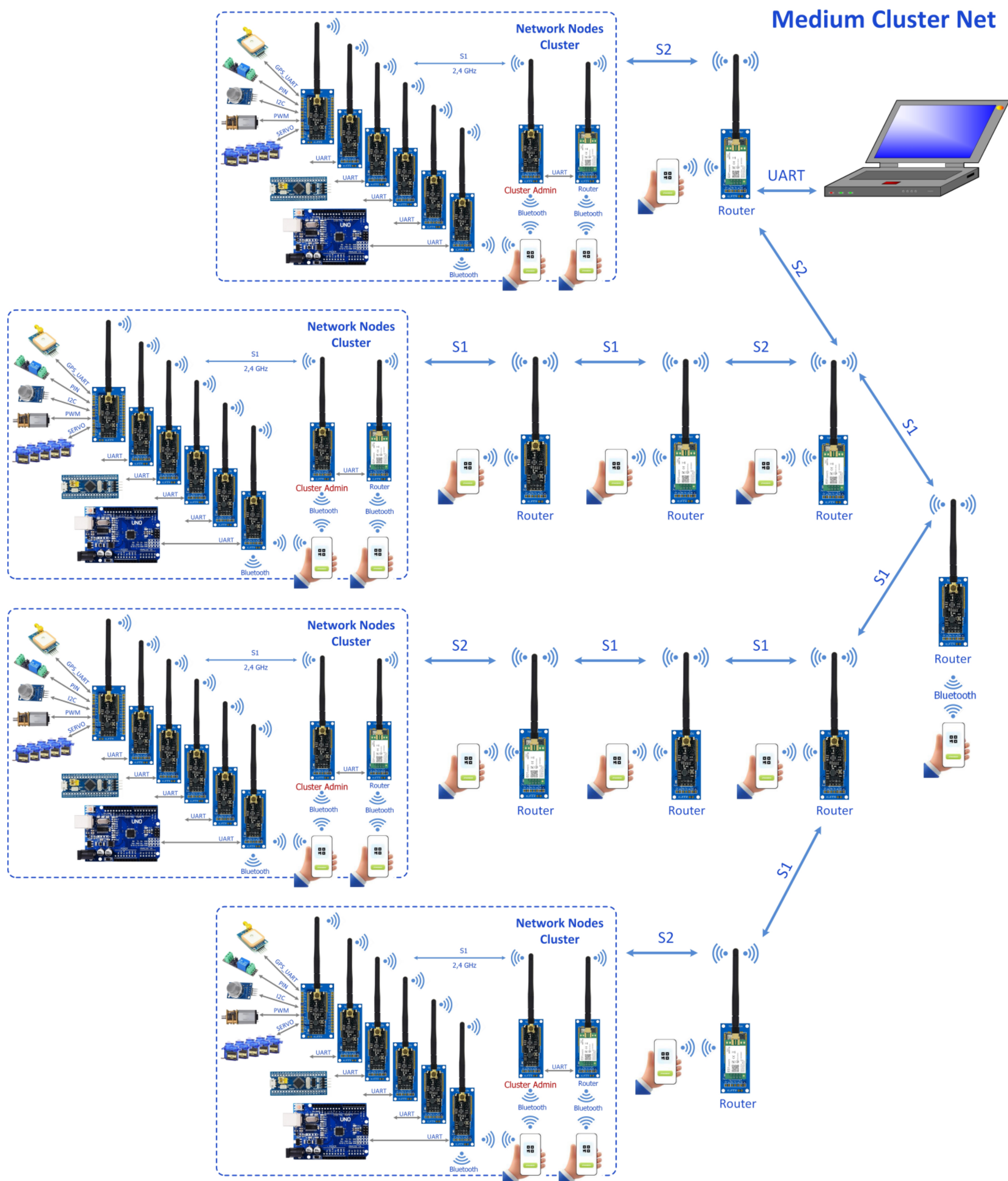


Рисунок 22 - Фізична реалізація мережі "Medium Cluster Net"

Модель "Union Cluster Net" об'єднує декілька локальних мереж різної архітектури в єдину систему. Використання мереж Internet та GSM дає

можливість побудови глобальної мережі Easy Net Everywhere з урахуванням певних обмежень.

Архітектура мережі на основі моделі "Union Cluster Net" представлена на рис. 23.

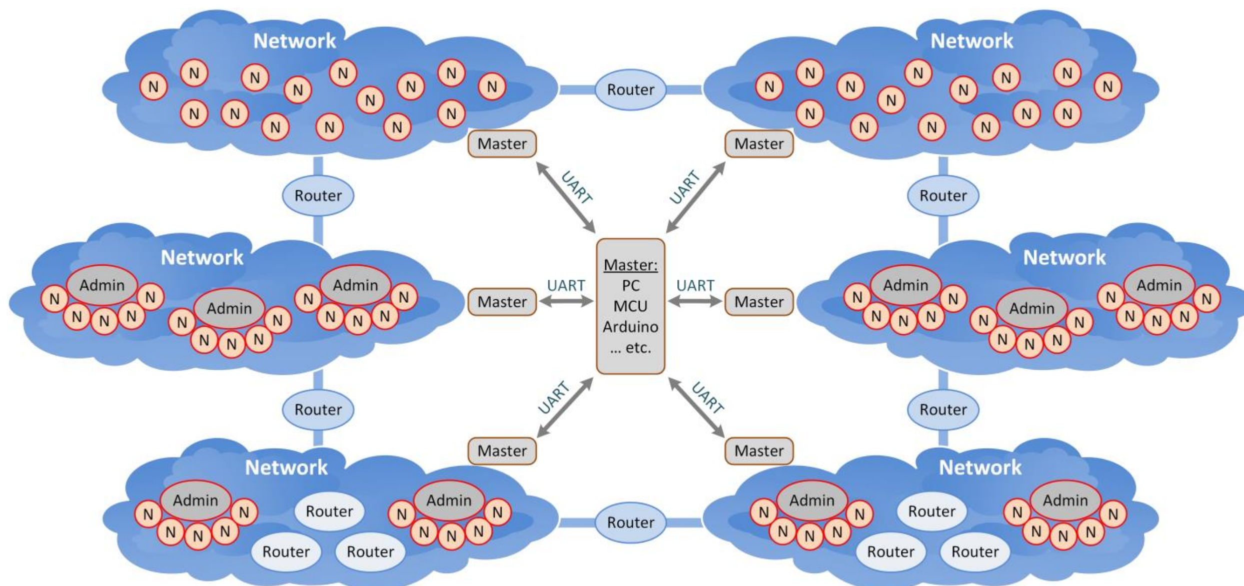


Рисунок 23 - Архітектура мережі на основі моделі "Union Cluster Net"

Адресація вузлів у мережі

Кожен вузол мережі має свою унікальну адресу. Повна адреса вузла складається з трьох компонентів: адреси мережі, адреси кластера і номера вузла в кластері.

Довжина адреси мережі становить 1 байт і адресує 254 мережі.

Довжина адреси кластера становить 4 біти і адресує 14 кластерів.

Довжина номера вузла становить 4 біти і адресує 15 вузлів.

Формат мережевої адреси наведено на рис. 24.

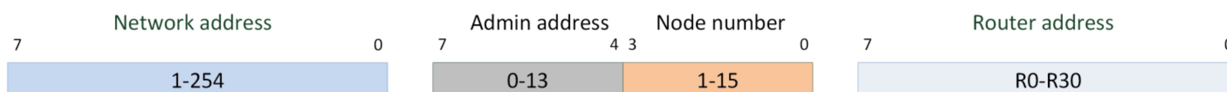


Рисунок 24 - Формат мережевої адреси

Поділ адресного простору мережі

Адресний простір мережі розділено на сегменти за функціональною ознакою пристроїв:

- вузли кластера мають номери: 1-15/0x01-0x0F;
- адміністратори кластера займають діапазон адрес: 0.0-13.0/0x00-0xD0;
- роутери та репітери займають діапазон адрес: R0-R30/0xE0-0xFE/.

Поділ адресного простору мережі на сегменти представлено на рис. 25.

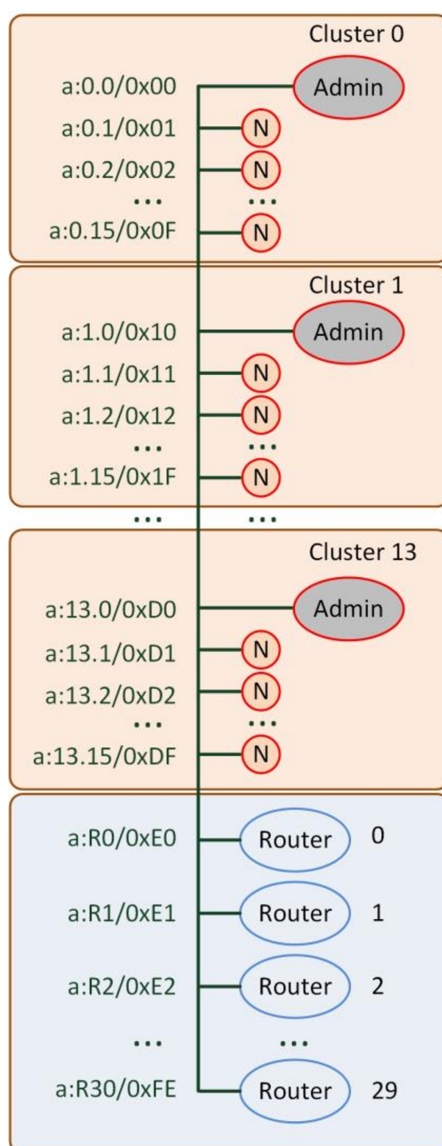


Рисунок 25 - Поділ адресного простору мережі на сегменти

Адреси вузлів мережі наведені в таблиці 2.

Таблиця 2 - Адреси вузлів мережі

Cluster Admin address		**Node address**		**Full Node address**		**Router address**			
alias:	hex:	alias:	hex:	alias:	hex:	alias:	hex:	alias:	hex:
0	0x00	1	0x01	0.1-1.15	0x01-0x0F	R0	0xE0	R16	0xF0
1	0x10	2	0x02	1.1-1.15	0x11-0x1F	R1	0xE1	R17	0xF1
2	0x20	3	0x03	2.1-2.15	0x21-0x2F	R2	0xE2	R18	0xF2
3	0x30	4	0x04	3.1-3.15	0x31-0x3F	R3	0xE3	R19	0xF3
4	0x40	5	0x05	4.1-4.15	0x41-0x4F	R4	0xE4	R20	0xF4
5	0x50	6	0x06	5.1-5.15	0x51-0x5F	R5	0xE5	R21	0xF5
6	0x60	7	0x07	6.1-6.15	0x61-0x6F	R6	0xE6	R22	0xF6
7	0x70	8	0x08	7.1-7.15	0x71-0x7F	R7	0xE7	R23	0xF7
8	0x80	9	0x09	8.1-8.15	0x81-0x8F	R8	0xE8	R24	0xF8
9	0x90	10	0x0A	9.1-9.15	0x91-0x9F	R9	0xE9	R25	0xF9
10	0xA0	11	0x0B	10.1-10.15	0xA1-0xAF	R10	0xEA	R26	0xFA
11	0xB0	12	0x0C	11.1-11.15	0xB1-0xBF	R11	0xEB	R27	0xFB
12	0xC0	13	0x0D	12.1-12.15	0xC1-0xCF	R12	0xEC	R28	0xFC
13	0xD0	14	0x0E	13.1-13.15	0xD1-0xDF	R13	0xED	R29	0xFD
Address range		15	0x0F	Address range		R14	0xEE	R30	0xFE
0.0-13.0	0x00-0xD0			0.1-13.15	0x01-0xDF	R15	0xEF		

Приклад адресації вузлів мережі

Якщо вузли належать до однієї мережі, то адреса мережі не вказується.

Приклад адресації вузлів мережі наведено на рис. 26.

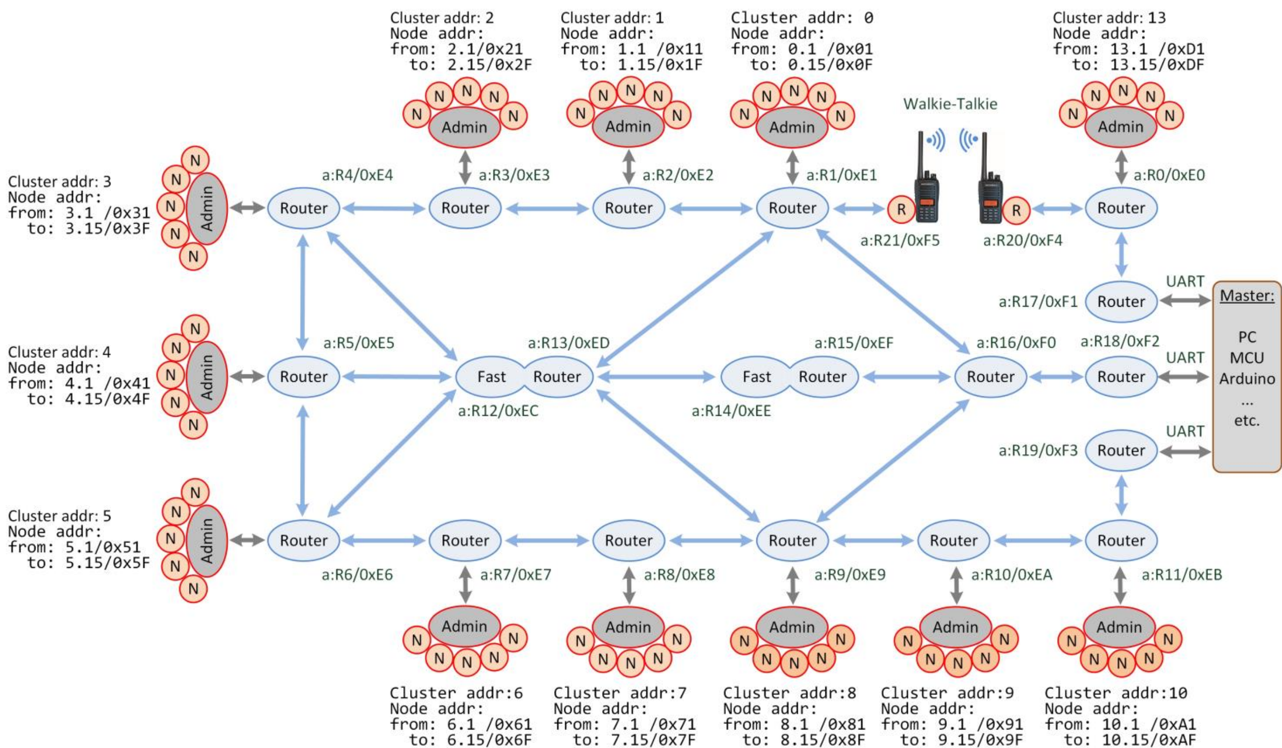


Рисунок 26 - Приклад адресації вузлів мережі

5. ПІДГОТОВКА ВУЗЛІВ МЕРЕЖІ ДО РОБОТИ

Для підготовки вузла мережі до роботи необхідно під'єднати його до джерела живлення 5 В або 3,3 В, як зазначено на рисунках 15 і 16.

Для взаємодії користувача з вузлом через комп'ютер, через мікроконтролер або через плату Arduino, необхідно роз'єм UART вузла під'єднати до роз'єму UART відповідного пристрою і встановити швидкість 115200 baud.

Для підключення вузла до комп'ютера використовується будь-який конвертор UART-USB. До комп'ютера через USB-Hub можна підключити кілька вузлів мережі.

Після цього можна увімкнути живлення вузла і виконати відповідні налаштування вузла за допомогою будь-якої термінальної програми.

Налаштування вузла можна виконати через планшет/смартфон за допомогою програми "Serial Bluetooth Terminal" (автор Kai Morich).

Для цього достатньо з'єднатися з вузлом по інтерфейсу Bluetooth (BLE) і виконати відповідні налаштування.

Цю процедуру необхідно виконати для кожного вузла мережі.

Перше ввімкнення вузла

Якщо всі підключення виконано правильно, то під час увімкнення живлення у вікні терміналу з'явиться таке повідомлення:

```
> Node 123 ready to start. Wait...
***********************
*  Easy Net EveryWhere
*  -------------------
*  Network.mode:   SIMPLE_NET
*  Radio.bitrate:  250Kb
*  UART.baud:      115200
*  Bluetooth:      ON 123 NET_NODE
***********************
>> Controller 123 started
```

Це означає, що вузол працездатний і готовий до роботи. Після цього необхідно виконати налаштування параметрів вузла.

Для налаштування вузла необхідно скористатися АТ-командами, які можуть бути надіслані вузлу з вікна терміналу або з планшета/смартфона.

АТ-команди виконують функції інтерфейсу пристрою з користувачем, що дає змогу програмі керування пристроєм інкапсулювати виклики системних функцій.

Програма "Serial Bluetooth Terminal". Базові відомості

Адміністрування мережі, обмін інформацією в мережі та налаштування параметрів вузлів мережі здійснюється не тільки через порт UART комп'ютера/мікроконтролера, а й через планшет/смартфон за допомогою програми "Serial Bluetooth Terminal" (автор Kai Morich).

Примітка: Користувач може використовувати будь-яку іншу аналогічну програму.

Програма завантажується з Internet і встановлюється на смартфон/планшет за допомогою програми Google Play.

Програма "Serial Bluetooth Terminal" дає змогу виконувати завдання:

1. Керування доступом до модуля Bluetooth (BLE) вузла мережі.
2. Встановлення режимів і параметрів вузла мережі за допомогою АТ-команд.
3. Приймання та відображення даних від вузла мережі.
4. Передавання даних вузлу мережі.
5. Адміністрування мережі.

Базовий набір функціональних кнопок

Програма "Serial Bluetooth Terminal" використовується для керування кількома різними пристроями за допомогою функціональних кнопок. Користувач може привласнити кожній кнопці ім'я і поставити у відповідність імені певну команду, яка відсилається керованому пристрою під час натискання кнопки. Кожен пристрій виконує свої функції зі своїми параметрами. Тому для керування кожним пристроєм необхідно мати окремий набір кнопок із командами керування, що не завжди зручно.

З метою спрощення інтерфейсу користувача, відповідно до парадигми: "Багато пристроїв - один інтерфейс", функціональні кнопки інтерфейсу називаються "F1", "F2" і т.д., а команди кнопок починаються з символу '~' (тільда).

Таким чином формується проксі-меню, яке в кожному конкретному пристрої має свою реалізацію, що дає змогу без зміни інтерфейсу користувача керувати функціями різних пристроїв.

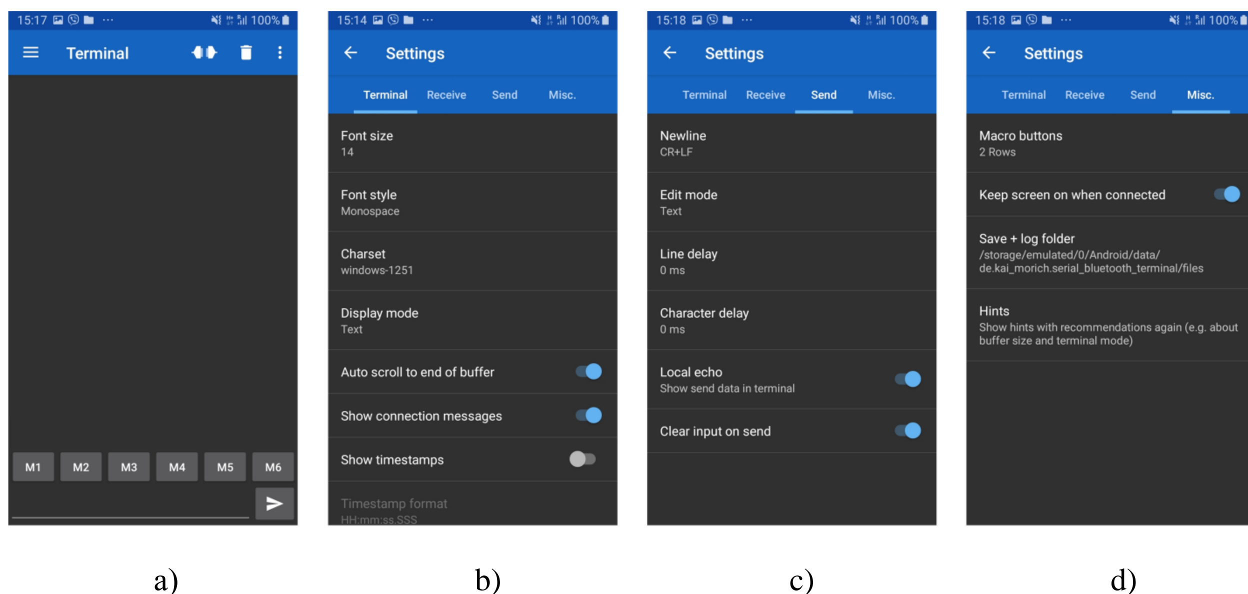
Базовий набір функціональних кнопок містить кнопки керування доступом до модуля Bluetooth (BLE) пристрою.

Розширений набір функціональних кнопок

Якщо базового набору функціональних кнопок недостатньо, то користувач створює додаткові кнопки, присвоює їм імена і ставить їм у відповідність виклики AT-команд.

Налаштування програми "Serial Bluetooth Terminal"

Після першого ввімкнення програми з'явиться вікно з набором кнопок, як показано на рис. 29.a. Користувачеві необхідно налаштувати інтерфейс взаємодії програми з вузлами мережі. Для цього у вкладках "Terminal", "Send" і "Misc." встановити параметри програми, як показано на рис. 29.b, c, d.



a) b) c) d)

Рисунок 29 - Установка параметрів програми

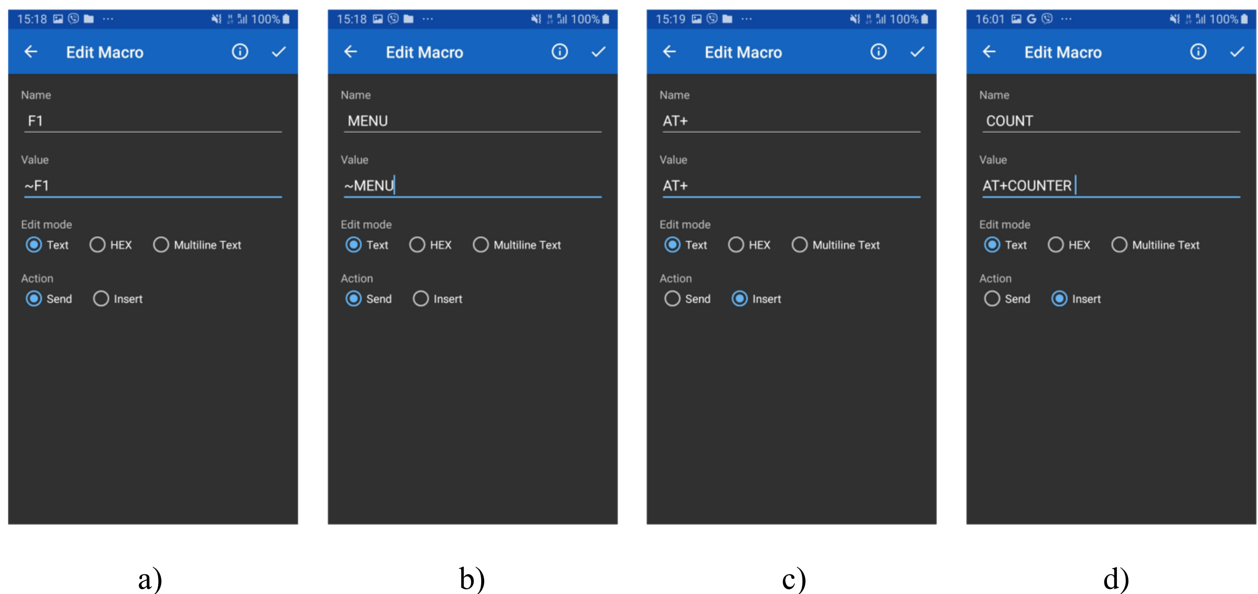
Потім необхідно присвоїти кнопкам імена і поставити їм у відповідність команди керування, як показано на рис. 30.a, b, c, d.

Порядок призначення імені кнопки і команди керування для неї:

1. Натискати на кнопку до появи вікна, як показано на рис. 30.a, b, c, d.

2. У верхньому рядку встановити ім'я кнопки, а в нижньому встановити команду керування.

Якщо команда не містить параметрів, то радіокнопку "Action" слід установити в положення "Send" (рис. 30.a, b), а якщо є параметри, то в положення "Insert" (рис. 30.c, d).



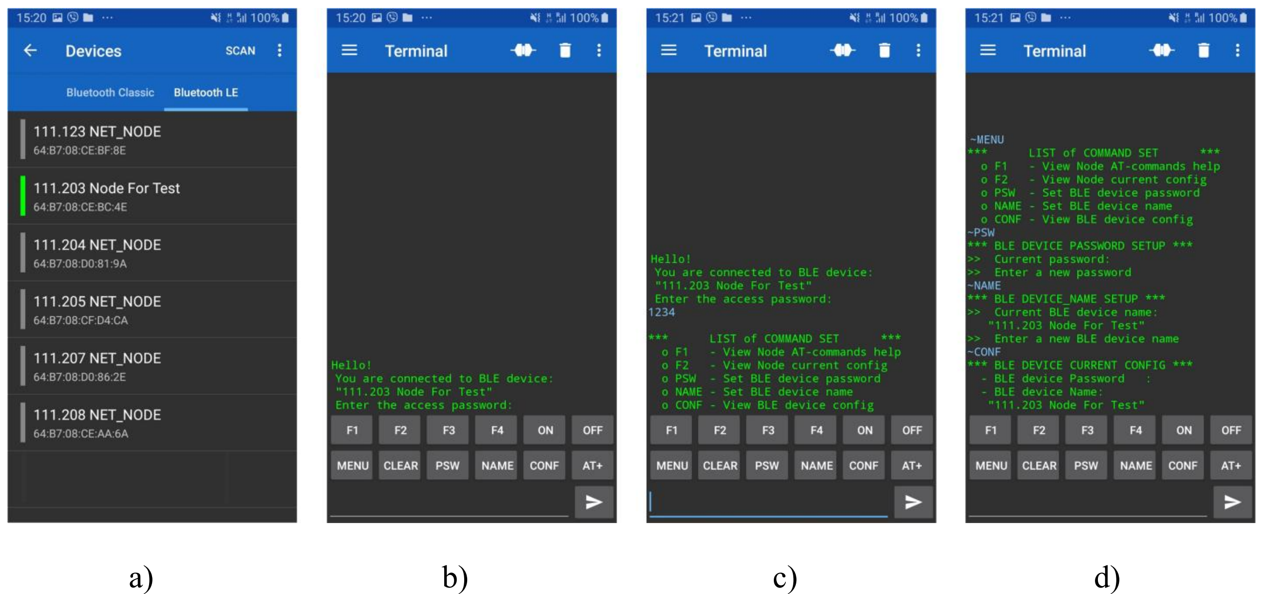
a) b) c) d)

Рисунок 30 - Присвоєння кнопкам імен і команд

Після присвоєння кнопці імені та команди необхідно натиснути галочку у верхньому правому куті екрана програми. Присвоїти імена та команди всім кнопкам, як показано на рис. 31.b.

Під час натискання на вкладку "Scan" програма просканує ефір і знайде пристрої Bluetooth (BLE) та відобразить на екрані у вигляді списку, як показано на рис. 31.a.

Для підключення до вузла мережі потрібно натиснути відповідний рядок у списку пристроїв.



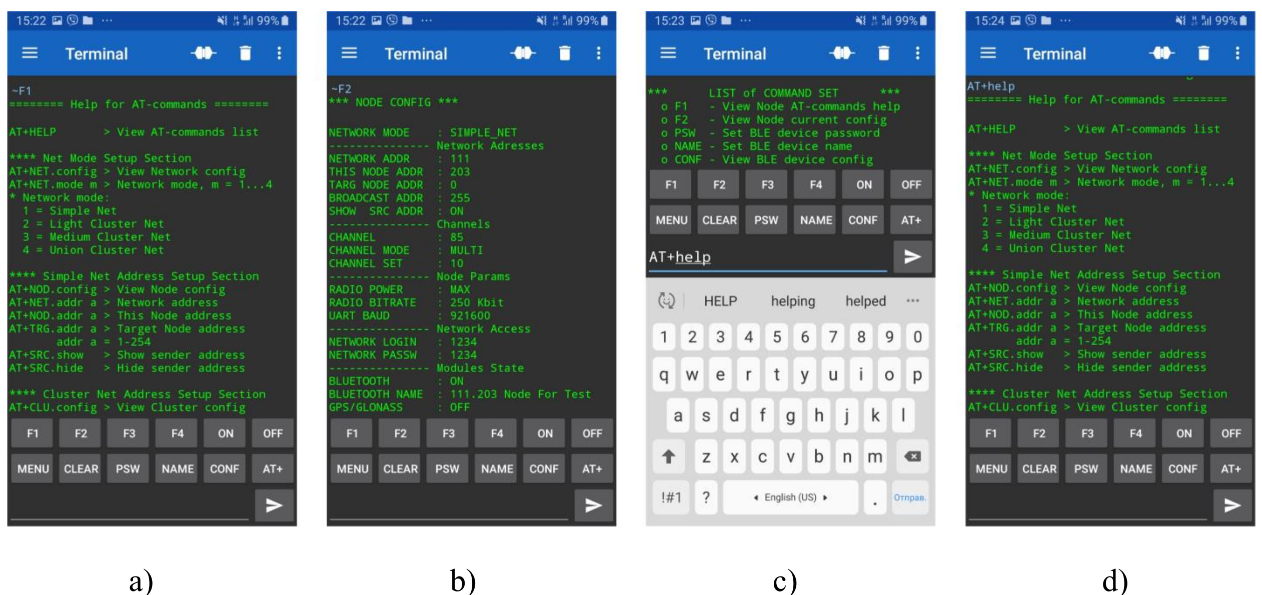
a) b) c) d)

Рисунок 31 - Приклад адресації вузлів мережі

У вікні термінала з'явиться повідомлення про під'єднання до вузла мережі та запит пароля доступу, як показано на рис. 31.b. Пароль за замовчуванням: "1234". Після введення пароля у вікні термінала з'явиться список кнопок і виконуваних команд, як показано на рис. 31.c.

На рис. 31.d показано виведення у вікно термінала меню командних кнопок.

На рис. 32.a і b показано виведення у вікно термінала результату натискання кнопок "F1" і "F2".



a) b) c) d)

Рисунок 32 - Виконання команд під час натискання кнопок

AT-команди мають префікс "AT+", який для зручності введення присвоєно відповідній кнопці. При натисканні кнопки "AT+" у рядку введення з'явиться префікс "AT+".

Для формування команди необхідно ввести частину AT-команди, що залишилася (на Приклад: "AT+HELP"), як показано на рис. 32.c і натиснути кнопку " > ".

Результат виконання AT-команди "AT+HELP" представлено на рис. 32.d. Аналогічним чином працюють усі інші командні кнопки.

Примітка:

Описані вище налаштування командних клавіш мають рекомендаційний характер.

Користувач на свій розсуд може встановлювати імена і команди для наявних командних кнопок, а також створювати додаткові кнопки для побудови зручного інтерфейсу керування вузлами мережі.

6. ТЕСТУВАННЯ МЕРЕЖІ. ПЕРЕДАЧА КОМАНД І ДАНИХ

Для перевірки правильності роботи командних кнопок і працездатності вузлів мережі необхідно під'єднати вузли відповідно до рис. 33. На комп'ютері має бути встановлена будь-яка термінальна програма.

Рисунок 33 - Підключення вузлів мережі для тестування

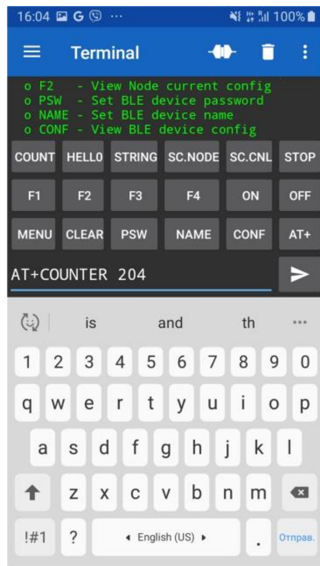
Тест якості зв'язку між двома вузлами запускається кнопкою "COUNT", яка передає вузлу AT- команду "AT+COUNTER target[,period]". Слід врахувати, що AT-команда має обов'язковий параметр, який необхідно ввести, і необов'язковий. Його можна не вказувати.

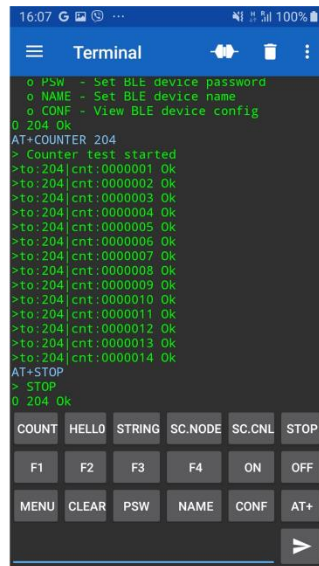
Обов'язковим параметром є адреса вузла-одержувача, до якого буде звернення.

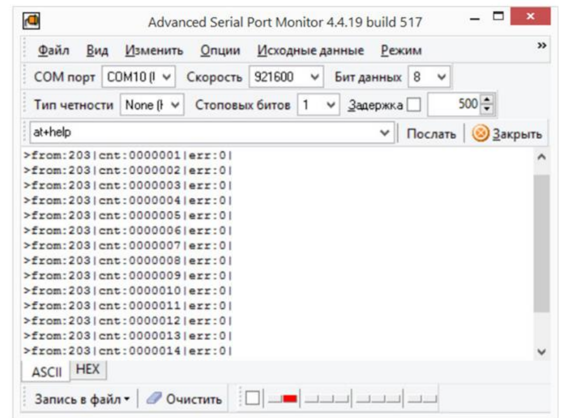
Приклад введення показано на рис. 34.a.

Після правильного введення команди та натискання кнопки ">" запуститься тест інкрементного лічильника, значення якого передаватиметься вузлу-одержувачу. Процес виконання тесту відображатиметься у вікні терміналу на смартфоні, як показано на рис.34.b і у вікні терміналу на комп'ютері (рис.34.c).

Кількість помилок слугує критерієм для визначення якості зв'язку на обраному каналі.







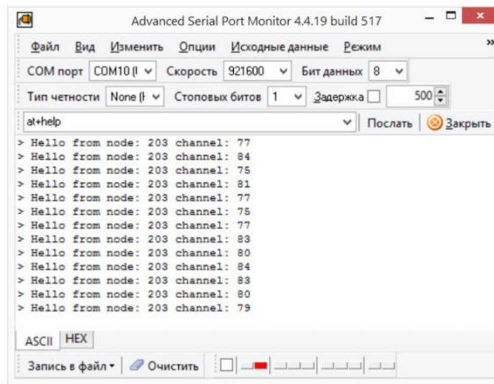
a) b) c)

Рисунок 34 - Запуск і виконання тестової команди "AT+COUNTER target[,period]"

Для тестування пропускної здатності каналу використовується тест "HELLO". Тест запускається на кількох вузлах, які надсилають повідомлення "Hello from node" одному вузлу.

Після введення команди "AT+HELLO target[,period]" і натискання кнопки ">" запуститься тест. Процес виконання тесту відображатиметься у вікні терміналу на смартфоні, як показано на рис. 35.a і у вікні терміналу на комп'ютері (рис. 35.b). Кількість колізій, що виникають, слугує критерієм визначення пропускної здатності каналу.

На рис. 35.c відображено виведення у вікно термінала смартфона результату виконання команди сканування каналів "AT+CNL.scan". З результату сканування випливає, що зайнятими каналами є канали 0-34, а канали 37-126 можна використовувати. Символ " # " у списку каналів позначає головний канал мережі.



a) b) c)

Рисунок 35 - Виконання тестової команди "AT+HELLO target[,period]"

На рисунку 36 представлено виведення у вікно терміналу планшета повідомлень тестової команди "AT+HELLO target[,period]" у процесі виконання. У тесті беруть участь 6 вузлів.

Вузол 203 - одержувач, а вузли 204, 205, 206, 207, 208 - відправники.

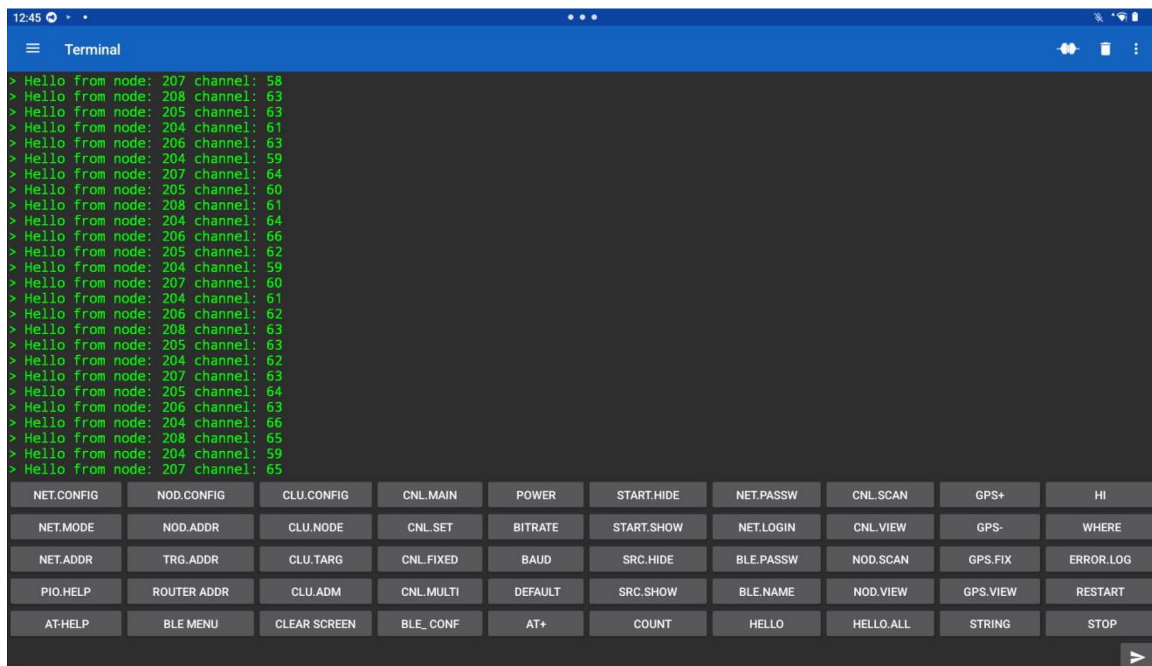


Рисунок 36 - Виконання тестової команди "AT+HELLO target[,period]" на 6 вузлах

На рисунку 37 представлено виведення повідомлень того самого тесту у вікна терміналів на комп'ютері через порти UART.

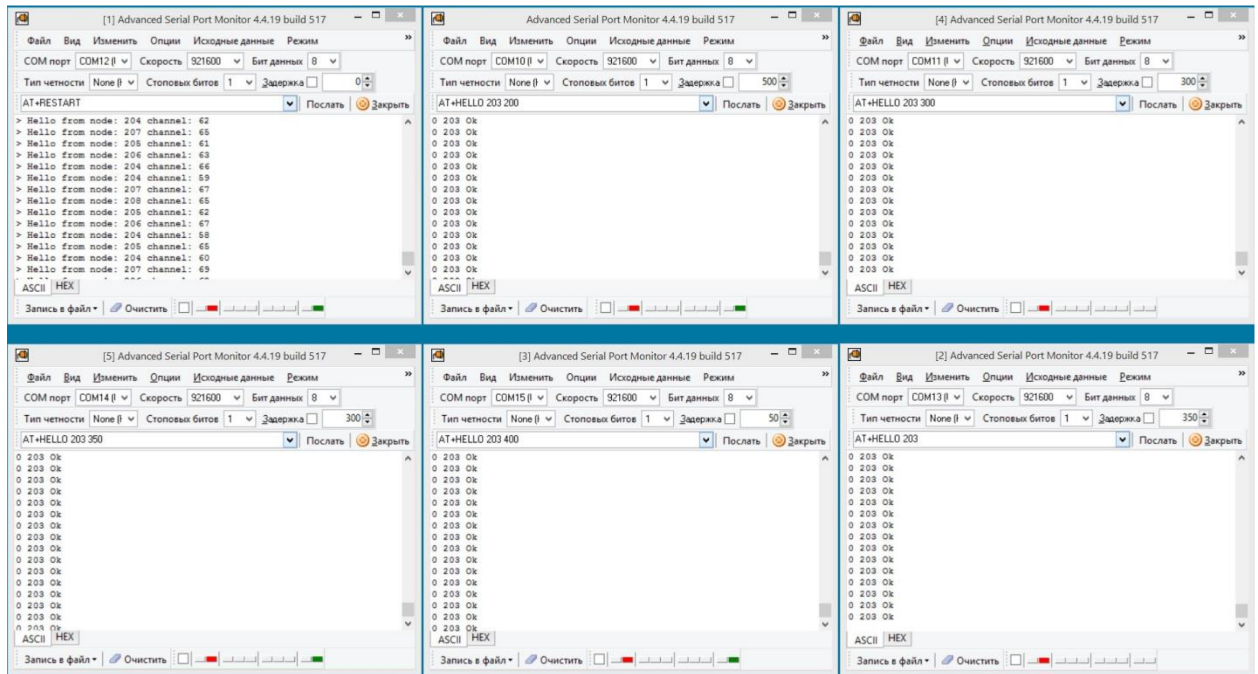


Рисунок 37 - Виконання тестової команди "AT+HELLO target[,period]" на 6 вузлах

Примітка:

Рядок "0 203 Ok" у вікні термінала (рис. 35) позначає:

0 - код квитанції успішної доставки пакета;

203 - адреса вузла-одержувача;

Ok - символьний еквівалент коду квитанції.

Рекомендація:

Перед початком роботи в мережі рекомендується встановити такі основні параметри:

Режим роботи мережі: команда "AT+NET.mode m";

Для всіх режимів мережі встановити:

Адреса мережі: команда "AT+NET.addr a";

Номер каналу мережі: команда "AT+CNL.main n";

Швидкість порту UART: команда "AT+BAUD bd";

Якщо обрано режим роботи мережі "Simple Net", то встановити:

Адреса вузла: команда "AT+NOD.addr a";

Якщо обрано режим роботи мережі "Light Cluster Net", додатково встановити:

Номер адміна кластера: команда "AT+CLU.adm n";

Номер вузла кластера: команда "AT+CLU.node n".

Якщо обрано режим роботи мережі "Medium Cluster Net", додатково встановити:

Адреси роутерів: команда "AT+ROUT.num n".

Адреси репітерів можна встановлювати в будь-якому режимі роботи мережі.

Подивитися поточні параметри вузла можна за допомогою команди: "AT+NOD.config"

Підключення обладнання для роботи в мережі

Підключення обладнання та організація процесу обміну даними в мережі є простим завданням.

Усі пристрої підключаються до вузлів мережі через послідовний інтерфейс UART відповідно до рис. 38 и 39.

Комп'ютери та контролери можна підключати в будь-якій комбінації. Приклад під'єднання комп'ютерів до вузлів мережі наведено на рис. 36.

Рисунок 38 - Підключення комп'ютерів до вузлів мережі

Приклад підключення контролерів до вузлів мережі подано на рис. 37.

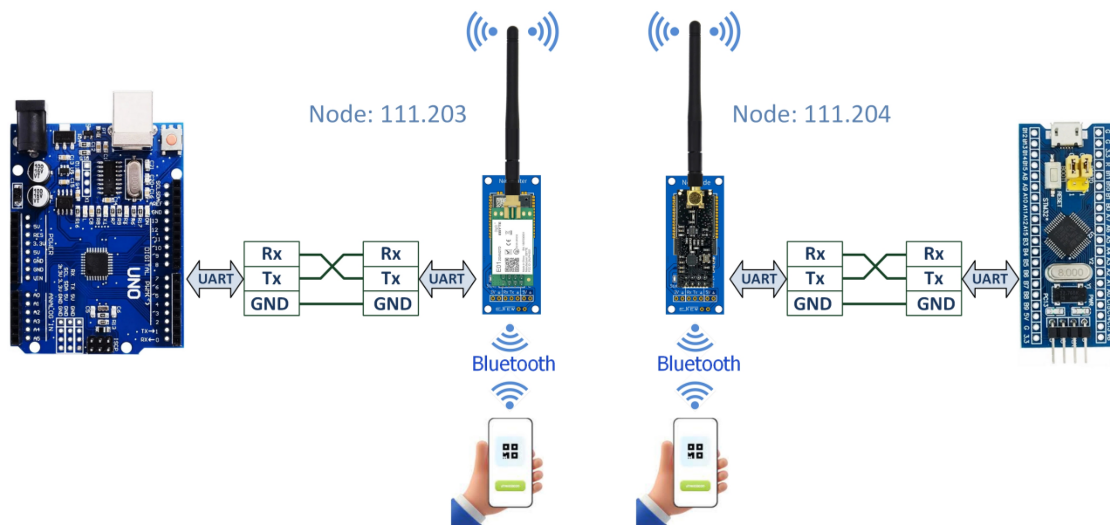


Рисунок 39 - Підключення контролерів до вузлів мережі

Якщо параметри вузлів мережі встановлено правильно, то мережа починає функціонувати. Перед початком експлуатації мережі рекомендується:

1. Просканувати канали та встановити номер головного каналу в середині найбільшої ділянки доступних каналів.

2. Виконати тестування за допомогою набору вбудованих тестів.

7. ОБМІН ДАНИМИ В МЕРЕЖІ. ПрикладИ ВИКОРИСТАННЯ

Порядок проходження байтів у переданому повідомленні не змінюється. Вузол-одержувач приймає дані з мережі та виводить їх у порт UART у тому самому порядку, у якому вони надійшли в порт UART вузла-відправника.

Тобто для користувача процес передачі даних є прозорим, як показано на рис. 40.

Швидкість портів UART вузлів встановлюється користувачем з урахуванням особливостей приймальних і передавальних пристроїв.

На Приклад, швидкість порту UART вузла-відправника може бути 921600 baud, а швидкість порту UART вузла-приймача - 115200 baud.

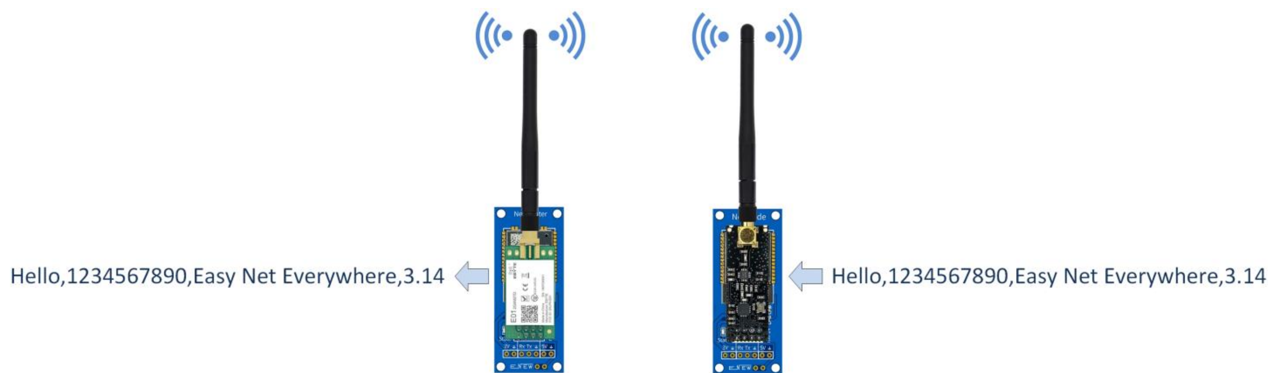


Рисунок 40 – Порядок проходження байтів у переданому повідомленні

Встановлення адреси вузла-одержувача

Перед надсиланням даних у мережі вузол-відправник має встановити адресу вузла одержувача в символьному або цифровому форматі. У цифровому форматі повідомлення більш компактне.

Встановлення адреси вузла-одержувача здійснюється трьома способами:

Спосіб 1: Використання AT-команди AT+TRG.addr a.

Адреса вузла-одержувача буде збережена в EEPROM. Усі наступні транзакції за замовчуванням здійснюватимуться тільки з цим вузлом.

Приклад:

```
AT+TRG.addr 203              //target address = 203
Hello! This is a simple message    /ag/messe to node 203
AT+CLU.targ 01.05            //target address:01.05(Cluster = 1 node = 5)
```

```
Hello! This is a simple message     //message to node 01.05
```

Спосіб 2: Встановлення адреси вузла-одержувача на початку повідомлення. Адреса в EEPROM не зберігається. Усі наступні транзакції будуть здійснюватися тільки з цим вузлом.

Приклад:

Символьний формат:

```
*203*Hello! This is a simple message      //message to node 203
*205*Hello! This is a simple message      //message to node 205
//
*01.05*Hello! This is a simple message    //message to node 01.05
*02.07*Hello! This is a simple message    //message to node 02.07
```

Цифровий формат:

```
#CB#Hello! This is a simple message       //message to node 203 = 0xCB
#CD#Hello! This is a simple message       //message to node 205 = 0xCD
//
#25#Hello! This is a simple message       //message to node 02.05
#27#Hello! This is a simple message       //message to node 02.07
```

Спосіб 3: Встановлення адреси вузла-одержувача в окремому повідомленні. Адреса в EEPROM не зберігається. Усі наступні транзакції здійснюватимуться тільки з цим вузлом.

Приклад:

Символьний формат:

```
*203*                                     //target address = 203
Hello! This is a simple message           //message to node 203
//
*02.07*
Hello! This is a simple message           //message to node 02.07
```

Цифровой формат:

```
#CB#                                      //target address = 203/0xCB
Hello! This is a simple message           //message to node 203

#25#
Hello! This is a simple message           //message to node 02.05
```

Надсилання даних вузлу-одержувачу

Вузлу-одержувачу можуть бути надіслані будь-які дані в символьному та цифровому форматі.

Дані можуть бути відформатовані, якщо дані є командним рядком зі списком параметрів або бути простим текстовим рядком.

Завдання парсингу даних лежить на користувачеві. Вузол-одержувач виводить в UART дані в тому ж порядку, в якому вони були відправлені вузлом-відправником.

Приклад 1:

Передання вузлу 203 команди !XXXX! з параметрами в символьному форматі.

```
String DATA_OUT = "*203*!XXXX!12,12345,234567,345.678";
printf(DATA_OUT);                   //send data to network
```

Передача вузлу 01.05 команди !XXXX! із параметрами у символьному форматі.

```
String DATA_OUT = "*01.05*!XXXX!12,12345,234567,345.678";
printf(DATA_OUT);                   //send data to network
```

На стороні вузла-одержувача дані можна прийняти в такий спосіб:

```
#include "Parser.h"
enum package{_cmd,_parm_1,_parm_2,parm_3,_parm_4}; //data format
byte DATA_IN[150];
String cmd;
int8 var_8; int16 var_16;
int32 var_32;float var_fl;

//== input data parsing
parse_data(DATA_IN);

//== get parameters
cmd = get_token_string(_cmd);           // cmd = !XXXX!
var_8 = get_token_int8(_parm_1);        // var_8 = 12
var_16 = get_token_int16(_parm_2);      // var_16 = 12345
var_32 = get_token_int32(_parm_3);      // var_32 = 234567
var_fl = get_token_float(_parm_4);      // var_fl = 345.678
```

8. РОБОТА З ВУЗЛОМ "CONTROLLER"

Призначення виводів вузла "Controller"

На друкованій платі вузла "Controller" для зручності використання всі виводи (пін, pin) позначені у форматі логічних номерів. Кожен пін може виконувати кілька функцій.

Функції пінів і відповідність їхніх логічних номерів фізичним номерам наведено в табл. 2.

Таблиця 2 - Вузол "Controller". Функції пінів і відповідність їхніх логічних номерів фізичним номерам

LOG_PIN	GPIO	ADC/PIN	DAC/PIN	SPI	I^2C	PWM/PIN	SERVO/PIN	GPS
01	27	2/1				1/1	1/1	
02	26	2/2	2/2			2/2	2/2	
03	25	2/3	1/3			3/3	3/3	
04	33	1/4				4/4	4/4	
05	35*	1/5						
06	34*	1/6						
07	36*	1/7						Rx
08	02	2/8				5/8	5/8	Tx
09	15	2/9		CS		6/9	6/9	
10	13	2/10		MOSI		7/10	7/10	
11	12	2/11		MISO		8/11	8/11	
12	14	2/12		SCK		9/12	9/12	
13	22	2/13			SCL	10/13	10/13	
14	21	2/14			SDA	11/14	11/14	

* - піни тільки для читання.

Команди управління вузлом Controller оперують з пінами, які мають логічні номери 01-14.

Класи команд управління функціями вузла "Controller"

Робота вузла «Controller» полягає у виконанні AT-команд, що надходять із мережі/Bluetooth/Wi-Fi та передачі їх у порт UART. AT-команди для вузла "Controller" мають три класи: "GET", "SET" та "CANCEL".

AT-команди класу GET повідомляють вузол Controller про те, що потрібно передати в мережу дані, зазначені в команді.

AT-команди класу "SET" встановлюють режим роботи вузла "Controller", вказаний у команді.

AT-команди класу "CANCEL" скасовують режим роботи вузла "Controller", вказаний у команді.

AT-команди мають два формати: повний та короткий. Повний формат зручно використовувати при ручному введенні команд, а короткий - для передачі команди в мережу. Формати AT-команд описані у Додатку 1.

8.1. МОДУЛЬ PIO. КОМАНДИ УПРАВЛІННЯ

Управління логічним рівнем пінів PIO

Модуль PIO може встановлювати на пінах вузла стан логічної "1" або логічного "0", а також надсилати в мережу поточне логічне значення одного або декількох пінів вузла, зазначених у команді.

Задача: Вузлу 203 встановити пін 1 у стан "0", пін 4 у стан "1", пін 8 у стан "0":

```
AT-команда    [*addr*]AT+PIN.SET
              pin:state[,pin:state][,pin:state]…[,pin:state]
Повернення    addr result  list old_pin_state>new_pin_state
              result: 0 cmd_Ok |1 cmd_ Error
 Приклад      *203*AT+PIN.SET 1:0,4:1,8:0
              203 0 cmd_Ok 1:1>0,4:1>1,8:1>0  //результат виконання команди
              203 1 cmd_Error                 //команду не виконано
```

Задача: Отримати стан пінів 1, 4, 8 вузла 203:

```
AT-команда    [*addr*]AT+PIN.GET pin[,pin][,pin]…[,pin]
Повернення    addr pin:state[,pin:state][,pin:state]…[,pin:state]|cmd_ Error
 Приклад      *203*AT+PIN.GET 1,4,8
Повернення    203 1:0,4:1,8:0
              203                  //номер вузла-відправника
              1:0,4:1,8:0          //номер піна та його стан
              203 1 cmd_Error      //команду не виконано
```

Встановлення режиму передавання повідомлень про стан пінів PIO у разі зміни їхнього логічного стану

Вузол "Controller" може надсилати повідомлення іншому вузлу при кожній зміні логічного стану одного або декількох пінів. Повідомлення передається тому вузлу, який встановив режим.

Задача: Встановити для вузла 203 режим передавання повідомлення при зміні логічного стану пінів:

```
AT-команда    [*addr*]AT+PIN.SET_CHANGE_NOTIFY  pin[,pin][,pin]…[,pin]
Повернення    addr  0 cmd_Ok |1 cmd_ Error
 Приклад      *203*AT+PIN.SET_CHANGE_NOTIFY 1,4,8
```

```
Повернення    203 0 cmd_Ok          //команду виконано
              203 1 cmd_Error       //команду не виконано
```

Після встановлення режиму вузол буде відсилати в мережу повідомлення при кожній зміні логічного рівня зазначених пінів:

```
Приклад       203 1:0>1         //зміна логічного стану піна 1 з 0 на 1
              203 4:0>1         //зміна логічного стану піна 4 з 0 на 1
              203 4:1>0,8:0>1   //зміна логічного стану пінів 4 і 8
```

Відміна режиму передавання повідомлень про стан пінів PIO у разі зміни їхнього логічного стану

Задача: Відмінити для вузла 203 режим передавання повідомлення у разі зміни логічного стану пінів:

```
AT-команда    [*addr*]AT+PIN.CANCEL_CHANGE_NOTIFY
Повернення    addr 0 cmd_Ok |1 cmd_ Error
Приклад       *203*AT+PIN.CANCEL_CHANGE_NOTIFY
Повернення    203 0 cmd_Ok | 203 1 cmd_Error
```

Встановлення режиму передавання повідомлень про стан пінів PIO із зазначеним періодом

Вузол "Controller" може надсилати повідомлення іншому вузлу про стан пінів PIO із заданим періодом period. Повідомлення передається тому вузлу, який встановив режим

Задача: Встановити для вузла 203 режим передавання повідомлень про стан пінів із зазначеним періодом. Період у мілісекундах 1-10000 або в секундах 1-1000:

```
AT-команда    [*addr*]AT+PIN.SET_PERIOD_STATE_NOTIFY_MS
              period,pin[,pin][,pin]…[,pin]
              [*addr*]AT+PIN.SET_PERIOD_STATE_NOTIFY_SEC
              period,pin[,pin][,pin]…[,pin]
Повернення    addr 0 cmd_Ok | 1 cmd_ Error
Приклад       *203*AT+PIN.SET_PERIOD_STATE_NOTIFY_MS 300,1,4,8 //період:300 ms
              *203*AT+PIN.SET_PERIOD_STATE_NOTIFY_SEC 600,1,4,8//період:10
              min.
Повернення    203 0 cmd_Ok | 203 1 cmd_Error
```

Після встановлення режиму вузол надсилатиме в мережу повідомлення про стан пінів із періодом 300 ms/10 min:

```
Приклад       203 1:0,4:1,8:0
              203 1:0,4:1,8:0
              203 1:0,4:1,8:1
```

Відміна режиму передавання повідомлень про стан пінів PIO із зазначеним періодом

Задача: Відмінити для вузла 203 режим передавання повідомлень про стан пінів із зазначеним періодом:

```
AT-команда    [*addr*]AT+PIN.CANCEL_PERIOD_STATE_NOTIFY
Повернення    addr 0 cmd_Ok | 1 cmd_ Error
Приклад       *203*AT+PIN.CANCEL_PERIOD_STATE_NOTIFY
Повернення    203 0 cmd_Ok | 203 1 cmd_Error
```

8.2. МОДУЛЬ ADC. КОМАНДИ УПРАВЛІННЯ МОДУЛЕМ

Контролер ESP32 має два канали ADC: ADC_1 та ADC_2. Канал ADC_2 можна використовувати тільки тоді, коли модуль Wi-Fi вимкнений.

Отримання значення напруги на пінах ADC

Модуль ADC може надсилати в мережу поточне значення в мілівольтах одного або декількох пінів вузла, зазначених у команді.

Задача: Отримати значення напруги на пінах 4,5,6,7 вузла 203:

```
AT-команда     [*addr*]AT+ADC.GET_ADC pin[,pin][,pin]…[,pin]
Повернення     addr pin:val[,pin:val][,pin:val]…[,pin:val] |cmd_ Error
 Приклад       *203*AT+ADC.GET_ADC 4,5,6,7
Повернення     203 4:105,5:2700,6:1350,7:3500
               4:105,5:2700,6:1350,7:3500 //напруга на входах піна в mv
               203 1 cmd_Error
```

Встановлення режиму передавання повідомлень у разі зміни рівня напруги на пінах ADC

Вузол "Controller" може надсилати повідомлення іншому вузлу при кожній зміні рівня напруги на вході одного або декількох пінів. Повідомлення передається тому вузлу, який встановив режим.

Задача: Встановити для вузла 203 режим передавання повідомлення в разі зміни рівня напруги на вході одного або декількох пінів у разі перевищення заданого порога **threshold**:

```
AT-команда     [*addr*]AT+ADC.SET_CHANGE_NOTIFY  threshold_mv,
               pin[,pin][,pin]…[,pin]
Повернення     addr  0 cmd_Ok |1 cmd_ Error
 Приклад       *203*AT+ADC.SET_CHANGE_NOTIFY 50,4,5,6,7 //порог 50 mv
Повернення     203 0 cmd_Ok | 203 1 cmd_Error
```

Після встановлення режиму вузол буде відсилати в мережу повідомлення при кожній зміні рівня напруги на вході зазначених пінів:

```
 Приклад       203 4:105>170 //зміна рівня напруги піна 4
               203 5:2700>21750 //зміна рівня напруги піна 5
               203 7:3500>3450 //зміна рівня напруги піна 7
```

Відміна режиму передавання повідомлень у разі зміни рівня напруги на пінах ADC

Задача: Відмінити для вузла 203 режим передавання повідомлення у разі зміни рівня напруги на пінах ADC:

AT-команда	`[*addr*]AT+ADC.CANCEL_CHANGE_NOTIFY`
Повернення	`addr 0 cmd_Ok |1 cmd_ Error`
Приклад	`*203*AT+ADC.CANCEL_CHANGE_NOTIFY`
Повернення	`203 0 cmd_Ok | 203 1 cmd_Error`

Встановлення режиму передавання повідомлень про рівні напруги на пінах ADC із заданим періодом

Вузол "Controller" може надсилати повідомлення про рівні напруги на пінах ADC іншому вузлу із заданим періодом **period**. Повідомлення передається тому вузлу, який встановив режим.

Задача: Встановити для вузла 203 режим передавання повідомлень про рівні напруги на пінах ADC із заданим періодом. Період у мілісекундах 1-10000 або в секундах 1-1000:

AT-команда	`[*addr*]AT+ADC.SET_PERIOD_VALUE_NOTIFY_MS period,pin[,pin][,pin]…[,pin]` `[*addr*]AT+ADC.SET_PERIOD_VALUE_NOTIFY_SEC period,pin[,pin][,pin]…[,pin]`
Повернення	`addr 0 cmd_Ok | 1 cmd_ Error`
Приклад	`*203*AT+ADC.SET_PERIOD_VALUE_NOTIFY_MS 300,4,5,7 //період:300 ms` `*203*AT+ADC.SET_PERIOD_VALUE_NOTIFY_SEC 600,4,5,7//період:10 min.`
Повернення	`203 0 cmd_Ok | 203 1 cmd_Error`

Після встановлення режиму вузол буде відсилати в мережу повідомлення про рівні напруги на пінах ADC з періодом 300 ms/10 min:

Пример

```
203 4:105   //рівень напруги на піні 4
203 5:2700  //рівень напруги на піні 5
203 7:3500  //рівень напруги на піні 7
```

Відміна режиму передавання повідомлень про рівні напруги на пінах ADC із заданим періодом

Задача: Відмінити для вузла 203 режим передавання повідомлень про рівні напруги на пінах ADC із зазначеним періодом:

AT-команда	[*addr*]AT+ADC.CANCEL_PERIOD_VALUE_NOTIFY
Повернення	addr 0 cmd_Ok | 1 cmd_ Error
Приклад	*203*AT+ADC.CANCEL_PERIOD_VALUE_NOTIFY
Повернення	203 0 cmd_Ok | 203 1 cmd_Error

8.3. МОДУЛЬ DAC. КОМАНДИ КЕРУВАННЯ МОДУЛЕМ

Вузол "Controller" має два піни DAC: DAC_1/PIN_3 і DAC_2/PIN_2. На пінах встановлюється значення заданої напруги з розрядністю 8 біт.

Встановлення рівня напруги на пінах DAC

Модуль DAC може встановлювати на пінах 2 і 3 заданий у команді рівень напруги в мілівольтах.

Задача: Вузлу 203 встановити на піні 2 напругу 1500 mv, а на піні 3 напругу 2500 mv:

AT-команда	[*addr*]AT+DAC.SET pin:value[,pin:value]
Повернення	addr 0 cmd_Ok | 1 cmd_ Error
Приклад	*203*AT+DAC.SET 2:1500,3:2500
	203 0 cmd_Ok | 203 1 cmd_Error

8.4. МОДУЛЬ PWM. КОМАНДИ УПРАВЛІННЯ МОДУЛЕМ

Вузол "Controller" має 11 пінів PWM. Для ініціалізації модуля PWM необхідно встановити такі параметри:

- частоту **frequency** у діапазоні 20-5000 Hz;
- розрядність **resolution** у діапазоні 8-12 bit;
- пін **pin** для виведення сигналу PWM (1 - 4; 8-14).

Управління роботою модуля PWM на зазначеному піні здійснюється встановленням параметра **duty** в діапазоні 0-1000, де 1000 = 100%.

Ініціалізація модуля PWM

Задача: Ініціалізувати модуль PWM вузла 203 з параметрами:

- **frequency** = 1000;
- **resolution** = 8;
- **pin** = 1,2,3.

AT-команда	[*addr*]AT+PWM.SET frequency,resolution,pin[,pin][,pin]…[,pin]
Повернення	addr 0 cmd_Ok | 1 cmd_ Error
Приклад	*203*AT+PWM.SET 1000,8,1,2,3
	203 0 cmd_Ok | 203 1 cmd_Error

Управління скважністю на пінах модуля PWM

Задача: На піні 1 вузла 203 встановити скважність 10%, на піні 2 - 25%, на піні 3 - 75%:

AT-команда	[*addr*]AT+PWM.DUTY pin:value[,pin:value][,pin:value]…[,pin:value]
Повернення	addr 0 cmd_Ok | 1 cmd_ Error
Приклад	*203*AT+PWM.DUTY 1:100,2:250,3:750
	203 0 cmd_Ok | 203 1 cmd_Error

Відміна режиму PWM

Задача: Відмінити для вузла 203 режим PWM

AT-команда	[*addr*]AT+PWM.CANCEL
Повернення	addr 0 cmd_Ok | 1 cmd_ Error

8.5. МОДУЛЬ SERVO. КОМАНДИ УПРАВЛІННЯ МОДУЛЕМ

Вузол "Controller" має 11 пінів для управління сервоприводами. Для ініціалізації модуля SERVO необхідно встановити такі параметри:

- пін **pin** для управління сервоприводом (1 - 4; 8-14).

Управління роботою модуля SERVO на зазначеному піні здійснюється встановленням параметрів:

- **degree** = 0 - 180; //кут установки вала сервопривода в градусах
- **time** = 0 - 10000 ms або 1-1000 sec. //час встановлення

Ініціалізація модуля SERVO

Задача: Ініціалізувати модуль SERVO вузла 203 для пінів 1,2,3,4:

AT-команда	[*addr*]AT+SERVO.SET pin[,pin][,pin]…[,pin]
Повернення	addr 0 cmd_Ok | 1 cmd_ Error
Приклад	*203*AT+SERVO.SET 1,2,3,4 203 0 cmd_Ok | 203 1 cmd_Error

Встановлення кута повороту вала сервоприводів на пінах модуля SERVO

Задача: Вал сервоприводу 1 встановити в положення 90 градусів,

Вал сервоприводу 2 встановити в положення 30 градусів за час 1500 ms,

Вал сервоприводу 3 встановити в положення 120 градусів за час 2000 ms,

Вал сервоприводу 4 встановити в положення 180 градусів за час 500 ms:

AT-команда	[*addr*]AT+SERVO.DEGREE pin:degree:time[,pin:degree:time]…[,pin:degree:time]
Повернення	addr 0 cmd_Ok | 1 cmd_ Error
Приклад	*203*AT+SERVO.DEGREE 1:900,2:300:1500,3:1200:2000,4:1800:500 203 0 cmd_Ok | 203 1 cmd_Error

Встановлення режиму сканування сервоприводів на пінах модуля SERVO

У режимі сканування вал сервоприводу циклічно переміщується від початкового значення кута повороту **beg_degree** до кінцевого значення кута

повороту **end_degree** за час **time** у секундах із зупинками в кінцевих точках на час **pause** у секундах.

Задача: Встановити режим сканування для двох сервоприводів на пінах 1 і 3 з параметрами:

Сервопривід 1: **beg_degree** = 0, **end_degree** = 1800, **time** = 10, **pause** = 0;

Сервопривід 2: beg_degree = 450, **end_degree** = 1350, **time** = 30, **pause** = 5:

AT-команда	`[*addr*]AT+SERVO.SET_SCAN pin:beg_degree:end_degree:time:pause…`
Повернення	`addr 0 cmd_Ok | 1 cmd_ Error`
Приклад	`*203*AT+SERVO.SET_SCAN 1:0:1800:10,2:450:1350:30:5` `203 0 cmd_Ok | 203 1 cmd_Error`

Відміна режиму сканування сервоприводів на пінах модуля SERVO

Задача: Відмінити для вузла 203 режим сканування

AT-команда	`[*addr*]AT+SERVO.CANCEL_SCAN`
Повернення	`addr 0 cmd_Ok | 1 cmd_ Error`

Відміна режиму SERVO

Задача: Відмінити для вузла 203 режим SERVO

AT-команда	`[*addr*]AT+SERVO.CANCEL`
Повернення	`addr 0 cmd_Ok | 1 cmd_ Error`

8.6. МОДУЛЬ GPS. КОМАНДИ КЕРУВАННЯ МОДУЛЕМ

Модуль GPS є зовнішнім модулем і під'єднується у відповідний роз'єм вузла "Controller".

Модуль GPS управляється за допомогою AT-команд.

Підключення модуля GPS

Задача: Підключити модуль GPS:

```
AT-команда    [*addr*]AT+GPS+
Повернення    addr  0 cmd_Ok | 1 cmd_ Error
 Приклад      *203*AT+GPS+
Повернення    203 0 cmd_Ok | 203 1 cmd_Error
```

Відключення модуля GPS

Задача: Відключити модуль GPS:

```
AT-команда    [*addr*]AT+GPS-
Повернення    addr  0 cmd_Ok | 1 cmd_ Error
 Приклад      *203*AT+GPS-
Повернення    203 0 cmd_Ok | 203 1 cmd_Error
```

Отримання поточних координат GPS

Задача: Отримати короткі поточні координати модуля GPS вузла 203:

```
AT-команда    [*addr*]AT+GPS.GET_COORDINATE
Повернення    ret_val | addr 0 cmd_Ok | 1 cmd_ Error
 Приклад      *203*AT+GPS.GET
Повернення    //Latitude,Longitude
              N48.09'47.6574",E17.08'11.2991"
              203 0 cmd_Ok | 203 1 cmd_Error
```

Отримання розширених координат модуля GPS

Задача: Отримати розширені координати модуля GPS вузла 203:

```
AT-команда    [*addr*]AT+GPS.GET_DATA
Повернення    ret_val | addr 0 cmd_Ok | 1 cmd_ Error
 Приклад      *203*AT+GPS.GET
Повернення    //Latitude,Longitude,Distance(m),Altitude(m),Speed(m/s)
              N48.09'47.6574",E17.08'11.2991",3500,160,3
              203 0 cmd_Ok | 203 1 cmd_Error
```

Фіксація поточних координат модуля GPS

Задача: Зафіксувати поточні координати модуля GPS вузла 203:

```
AT-команда    [*addr*]AT+GPS.FIX
Повернення    ret_val | addr 0 cmd_Ok | 1 cmd_ Error
 Приклад      *203*AT+GPS.GET
Повернення    //Latitude,Longitude
              N48.09'47.6574",E17.08'11.2991"
              203 0 cmd_Ok | 203 1 cmd_Error
```

Встановлення режиму передавання повідомлень у мережу повідомлення про зміну координат GPS

Вузол "Controller" може надсилати повідомлення іншому вузлу в разі зміни координат GPS, що перевищують вказане значення. Повідомлення передається тому вузлу, який встановив режим.

Задача: Встановити для вузла 203 режим передавання повідомлення при зміні координат GPS у разі перевищення заданого значення **threshold** (1-100 m):

```
AT-команда    [*addr*]AT+GPS.SET_CHANGE_NOTIFY  threshold
Повернення    addr  0 cmd_Ok |1 cmd_ Error
 Приклад      *203*AT+GPS.SET_CHANGE_NOTIFY 10  //порог = 10 метров
Повернення    203 0 cmd_Ok | 203 1 cmd_Error
```

Після встановлення режиму вузол надсилатиме в мережу повідомлення при кожній зміні координат GPS, що перевищують зазначене значення:

```
 Приклад      N48.09'47.6574",E17.08'11.2991"
              N48.09'47.6751",E17.08'11.5622"
              N48.09'47.6751",E17.08'11.7988"
              N48.09'47.6554",E17.08'12.0528"
```

Відміна режиму передавання повідомлень у мережу повідомлення про зміну координат GPS

Задача: Відмінити для вузла 203 режим передавання повідомлення у разі зміни координат GPS:

```
AT-команда    [*addr*]AT+GPS.CANCEL_CHANGE_NOTIFY
Повернення    addr 0 cmd_Ok |1 cmd_ Error
 Приклад      *203*AT+GPS.CANCEL_CHANGE_NOTIFY
```

```
Повернення    203 0 cmd_Ok | 203 1 cmd_Error
```

Встановлення режиму передавання повідомлень у мережу координат GPS із заданим періодом

Вузол "Controller" може надсилати повідомлення з координатами GPS іншому вузлу із заданим періодом **period**. Повідомлення передається тому вузлу, який встановив режим.

Задача: Встановити для вузла 203 режим передавання повідомлень із координатами GPS із заданим періодом. Період у мілісекундах 100-10000 (0.1-10 sec):

```
AT-команда    [*addr*]AT+GPS.SET_PERIOD_NOTIFY  period
Повернення    addr 0 cmd_Ok | 1 cmd_ Error
 Приклад      *203*AT+GPS.SET_PERIOD_NOTIFY 200   //период: 200 ms
Повернення    203 0 cmd_Ok | 203 1 cmd_Error
```

Після встановлення режиму вузол буде відсилати в мережу повідомлення з координатами GPS з періодом 200 ms:

```
 Приклад      N48.09'47.6574",E17.08'11.2991"
              N48.09'47.6574",E17.08'11.2991"
              N48.09'47.6574",E17.08'11.2991"
```

Відміна режиму передавання повідомлень у мережу координат GPS із заданим періодом

Задача: Скасувати для вузла 203 режим передавання повідомлень із координатами GPS із заданим періодом:

```
AT-команда    [*addr*]AT+GPS.CANCEL_PERIOD_NOTIFY
Повернення    addr 0 cmd_Ok | 1 cmd_ Error
 Приклад      *203*AT+GPS.CANCEL_PERIOD_NOTIFY
Повернення    203 0 cmd_Ok | 203 1 cmd_Error
```

Додаток 1

АТ-КОМАНДИ КЕРУВАННЯ ПАРАМЕТРАМИ ВУЗЛІВ МЕРЕЖІ

Команда «AT+HELP»

Призначення: показати список АТ-команд. Регістр введення не має значення.

Приклад:

```
AT+HELP
```

У вікні терміналу відобразиться список АТ-команд:

```
======== Help for AT-commands ========
AT+HELP       > View AT-commands list

**** Net Mode Setup Section
AT+NET.config > View Network config
AT+NET.mode m > Network mode, m = 1...4
  Network mode:
  1 = Simple Net
  2 = Light Cluster Net
  3 = Medium Cluster Net
  4 = Union Cluster Net

**** Simple Net Address Setup Section
AT+NOD.config > View Node config
AT+NET.addr a > Network address
AT+NOD.addr a > This Node address
AT+TRG.addr a > Target Node address
       addr a = 1-254
AT+SRC.show   > Show sender address
AT+SRC.hide   > Hide sender address

**** Cluster Net Address Setup Section
AT+CLU.config > View Cluster config
AT+CLU.adm n  > This Admin number:  0-13
AT+CLU.node n > This Node number:   1-15
AT+CLU.targ n > Target Node number: 1-15
AT+ROUT.num n > Router number, n:R0-R30

**** Channel Mode Setup Section
AT+CNL.main n > Main channel number
            n = 1-125
AT+CNL.fixed  > Use one fixed channel
AT+CNL.multi  > Use many channels
AT+CNL.set s  > Channel set, s:1-50
AT+CNL.scan   > Scan and view channels
AT+CNL.view   > View available channels
```

```
**** Node Params Setup Section
AT+POWER pow  > Radio power,
   pow:1/2/3  = MIN/AVR/MAX
AT+BITRATE br > Radio bitrate,
   br:1/2/3   = 250Kb/1Mb/2Mb
AT+BAUD bd    > UART baud,
   bd:9600/115200/230400/460800/921600
AT+UART.flow+ > UART flow control ON
AT+UART.flow- > UART flow control OFF

**** Node Control Section
AT+START.hide > Hide start message
AT+START.show > Show start message
AT+DEFAULT    > Reset to default settings
AT+RESTART    > Node restart

**** Network Access Setup
AT+LOGIN log  > Login,    log:1-8 chars
AT+PASSW psw  > Password, psw:1-8 chars

**** Network Service Section
AT+HI         > Flash BLE connected node
AT+WHERE node > Flash node looking for
AT+ERROR.log  > View Error log
AT+NOD.scan   > Scan and view Net nodes
AT+NOD.view   > View available Net nodes

**** GPS Module Control Section
AT+GPS+       > GPS switch ON
AT+GPS-       > GPS switch OFF
AT+GPS.fix    > Fix GPS coordinates
AT+GPS.view   > View GPS coordinates

**** BLE Module Control Section
AT+BLE+       > BLE switch ON
AT+BLE-       > BLE switch OFF
AT+BLE.config > View BLE config
AT+BLE.name n > Set BLE name, n:name
AT+BLE.psw p  > Set BLE password, p:passw

****Test Mode Section
AT+STOP       > Test stop
AT+COUNTER target[,period]
  target   = target_addr,
  period   = 50...10000 ms, default: 500

AT+HELLO target[,period]
  target   = target_addr,
  period   = 50...10000 ms, default: 500
```

```
AT+HELLO.all [period]
  period   = 50...10000 ms, default: 500

AT+STRING target[,period][,length]
  target  = target_addr,
  period  = 50...1000 ms, default: 500
  length  = 1...150 bytes, default: 50

**** Message Format Section
*target addr* message 1...256 bytes
```

******** Net Mode Setup Section**

Команда «AT+NET.config»

Призначення: показати конфігурацію мережі. Буде відображено список вузлів мережі відповідно до поточної моделі мережі.

Приклад:

```
AT+NET.config
```

У вікні терміналу відобразиться конфігурація мережі:

```
===== Network Configuration =====

> CLUSTER   ADDR    NODE_TYPE   STATE
  0         0.0     Admin       ON
  0         0.1     Lite Node   ON
  0         0.2     Lite Node   ON
  0         0.3     Controller  ON
  0         0.4     Controller  ON
  0         0.5     Controller  ON
  0         0.6     Lite Node   OFF
  ----------------------------
> CLUSTER   ADDR    NODE_TYPE   STATE
  1         1.0     Admin       ON
  1         1.1     Controller  ON
  1         1.2     Controller  ON
  1         1.3     Controller  ON
  1         1.4     Controller  ON
  1         1.5     Controller  ON
  ----------------------------
> ROUTER    ADDR    NODE_TYPE   STATE
  1         R0      Single      ON
  2         R1/R2   Fast        ON
  3         R3/R4   Fast        ON
  4         R5      Single      ON
  ----------------------------
> REPEATOR  ADDR   TYPE         STATE
  1         R17    Repeator     OFF
```

Команда «AT+NET.mode m»

Призначення: встановити конфігурацію мережі.

Параметр m набуває значень:

1 = Simple Net

2 = Light Cluster Net

3 = Medium Cluster Net

4 = Union Cluster Net

Приклад:

```
AT+NET.mode 2
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> mode LIGHT_CLUSTER_NET: Ok
```

Якщо параметр AT-команд введено неправильно, то буде відображено повідомлення про помилку.

Приклад:

```
AT+NET.mode 5

> Error: Incorrect value: 5
```

******** Simple Net Addresses Setup Section**

Команда «AT+NOD.config»

Призначення: показати конфігурацію вузла.

Приклад:

AT+NOD.config

У вікні терміналу відобразиться поточна конфігурація вузла:

```
*** NODE CONFIG ***

NETWORK MODE     : SIMPLE_NET
--------------- Network Adresses
NETWORK ADDR     : 111
THIS NODE ADDR   : 203
TARG NODE ADDR   : 0
BROADCAST ADDR   : 255
```

```
SHOW  SRC ADDR  : ON
--------------- Channels
CHANNEL         : 85
CHANNEL MODE    : MULTI
CHANNEL SET     : 10
--------------- Node Params
RADIO POWER     : MAX
RADIO BITRATE   : 250 Kbit
UART BAUD       : 921600
--------------- Network Access
NETWORK LOGIN   : 1234
NETWORK PASSW   : 1234
--------------- Modules State
BLUETOOTH       : ON
BLUETOOTH NAME  : 111.203 Node For Test
GPS/GLONASS     : OFF
```

Команда «AT+NET.addr a»

Призначення: присвоїти адресу мережі.

Параметр а приймає значення: 1-254.

Приклад:

```
AT+NET.addr 234
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Network Address 234: Ok
```

Команда «AT+NOD.addr a»

Призначення: присвоїти адресу вузлам "Net Master", "Controller" і "Light Node".

Параметр **а** приймає значення: 1-254.

Приклад:

```
AT+NOD.addr 204
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> This Node Address 204: Ok
```

Команда «AT+TRG.addr a»

Призначення: присвоїти адресу вузлу-одержувачу.

Параметр **a** приймає значення: 1-254.

Приклад:

```
AT+ TRG.addr 127
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Target Node Address 127: Ok
```

Команда «AT+SRC.show»

Призначення: дозволяє виведення адреси вузла-відправника в UART вузла-приймача. Сервісна функція. Служить для візуальної ідентифікації користувачем вузла-відправника.

Приклад:

```
AT+SRC.show
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Sender address put to UART: Ok
```

Адреса вузла-відправника буде виводитися в UART вузла-приймача.

Команда «AT+SRC.hide»

Призначення: скасовує виведення в UART вузла-приймача адреси вузла-відправника.

Приклад:

```
AT+SRC.hide
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Sender address don't put to UART: Ok
```

Адреса вузла-відправника не буде виводитися в UART вузла-приймача.

******** Cluster Net Addresses Setup Section**

Команда «AT+CLU.config»

Призначення: показати конфігурацію поточного кластера.

Приклад:

```
AT+CLU.config
```

У вікні терміналу відобразиться поточна конфігурація кластера:

```
> Cluster Admin address: 08.00
   Node addresses:
   08.01
   08.02
   08.03
   08.04
```

Команда «AT+CLU.adm n»

Призначення: присвоїти номер вузлу "Cluster Admin".

Параметр **n** приймає значення: 1-13.

Приклад:

```
AT+CLU.adm 12
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> This Cluster Admin number 12: Ok
```

Команда «AT+CLU.node n»

Призначення: присвоїти номер вузлам кластера "Controller" або "Light Node".

Параметр **n** приймає значення: 1-15.

Приклад:

```
AT+CLU.node 12.15
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> This Cluster Node number 12.15: Ok
```

Команда «AT+CLU.targ n»

Призначення: присвоїти номер вузлам кластера-одержувача "Controller" або "Light Node".

Параметр **n** приймає значення: 0-13.1-15.

Приклад:

```
AT+CLU.targ 12.15
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Target Cluster Node number 12.15: Ok
```

Команда «AT+ROUT.num n»

Призначення: присвоїти номер вузлу "Net Router".

Параметр **n** приймає значення: R0-R30.

Приклад:

```
AT+ AT+ROUT.num R15
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> This Router number 15: Ok
```

******** Channel Mode Setup Section**

Команда «AT+CNL.main n»

Призначення: встановити для мережі номер основного каналу.

Параметр **n** приймає значення: 1-125.

Приклад:

```
AT+CNL.main 85
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Main channel #85: Ok
```

Команда «AT+CNL.fixed»

Призначення: встановити для мережі один фіксований номер каналу.

Приклад:

```
AT+CNL.fixed
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Fixed channel mode: Ok
```

Команда «AT+CNL.multi»

Призначення: дозволити вузлам мережі використовувати безліч каналів.

Приклад:

```
AT+CNL.multi
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Multi channels mode: Ok
```

Команда «AT+CNL.scan»

Призначення: сканувати і відобразити всі канали мережі з метою визначення вільних і зайнятих каналів.

Приклад:

```
AT+CNL.scan
```

У вікні терміналу відобразиться результат сканування каналів:

```
______________________________________________________________________________________________________________________________
 Channels State for Easy Net EveryWhere Environment. Legend: X - busy channel. Free channels:55 of 125

 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX                                    XX
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX                                    XX
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX                                   XXX
....:....|....:....|....:....|....:....|....:....|....:....|....:....|....:....|....:....|....:....|....:....|....:....|....:
0   5    10   15   20   25   30   35   40   45   50   55   60   65   70   75   80   85   90   95   100  105  110  115  120  125
```

Команда «AT+CNL.view»

Призначення: показати доступні канали.

Приклад:

```
AT+CNL.view
```

У вікні терміналу відобразиться список доступних каналів:

```
*> Available Channels:91

 31  35  37  38  39  40  41  42  43  44
 45  46  47  48  49  50  51  52  53  54
 55  56  57  58  59  60  61  62  63  64
 65  66  67  68  69  70  71  72  73  74
 75  76  77  78  79  80  81  82  83  84
 86  87  88  89  90  91  92  93  94  95
 96  97  98  99 100 101 102 103 104 105
106 107 108 109 110 111 112 113 114 115
116 117 118 119 120 121 122 123 124 125
126
```

Команда «AT+CNL.set s»

Призначення: виділити для мережі безліч робочих каналів s.

Параметр **s** приймає значення: 1-60.

Приклад:

```
AT+CNL.set 10
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Installed set of 10 channels: Ok
```

******** Node Params Setup Section**

Команда «AT+POWER pow»

Призначення: встановити потужність трансивера.

Параметр **pow** приймає значення: 1/2/3 = MIN/AVR/MAX.

Приклад:

```
AT+POWER 3
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> RF power MAX: Ok
```

Команда «AT+BITRATE br»

Призначення: встановити швидкість передавання даних трансивера.

Параметр br набуває значень: 1/2/3 = 250Kb/1Mb/2Mb.

Приклад:

```
AT+BITRATE 1
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно:

```
> RF bitrate 250 Kbit: Ok
```

Команда «AT+BAUD bd»

Призначення: встановити швидкість передавання даних порту UART.

Параметр bd приймає значення: 9600/115200/230400/460800/921600 baud.

Приклад:

```
AT+BAUD 921600
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> UART baud 921600: Ok
Don't forget to switch to installed baud
```

Команда «AT+UART.flow+»

Призначення: увімкнути керування потоком даних UART XON/XOFF. Відправник буде призупиняти передачу даних в UART при отриманні байта XOFF і відновлювати при отриманні байта XON.

Управління потоком даних запобігає переповненню вхідного буфера UART.

Приклад:

```
AT+UART.flow+
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно:

```
> UART flow control ON: Ok
```

Команда «AT+UART.flow-»

Призначення: вимкнути керування потоком даних UART XON/XOFF.

Приклад:

```
AT+UART.flow-
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно:

```
> UART flow control OFF: Ok
```

******** Node Control Section**

Команда «AT+START.hide»

Призначення: заборонити виведення повідомлення з параметрами вузла під час старту.

Приклад:

```
AT+START.hide
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Start message hide: Ok
```

Під час старту вузла у вікні термінала відобразиться повідомлення:

```
>> Node 203 started
```

Команда «AT+START.show»

Призначення: дозволити виведення повідомлення з параметрами вузла під час старту.

Приклад:

```
AT+START.show
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Start message show: Ok
```

Під час старту вузла у вікні термінала відобразиться повідомлення:

```
> Node 203 ready to start. Wait..
***********************
*  Easy Net EveryWhere
*  -------------------
*  Network.mode:   SIMPLE_NET
*  Radio.bitrate:  250Kb
*  UART.baud:      921600
*  Bluetooth:      ON
*  Bluetooth name: 111.203 Node For Test
***********************
>> Node 203 started
```

Команда «AT+DEFAULT»

Призначення: скинути налаштування параметрів вузла до початкових значень.

Приклад:

```
AT+DEFAULT
```

Вузол встановить параметри за замовчуванням.

Команда «AT+RESTART»

Призначення: перезавантажити вузол.

Приклад:

```
AT+RESTART
```

У вікні термінала відобразиться повідомлення про перезавантаження, після чого вузол буде перезавантажено.

```
> The node will reboot now
***********************
*  Easy Net EveryWhere
*  -------------------
*  Network.model:  SIMPLE_NET
*  Radio.bitrate:  250Kb
*  UART.baud:      115200
*  Bluetooth:      ON
*  Bluetooth name: 111.203 NET_NODE
***********************
      >> Node 203 started
```

******** Network Access Setup**

Команда «AT+LOGIN log»

Призначення: встановити мережевий логін.

Параметр **log** набуває значень: будь-які символи і цифри довжиною до 8 байт.

Приклад:

```
AT+LOGIN 12345678
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Network Login Ok: 12345678
```

Команда «AT+PASSW psw»

Призначення: встановити мережевий пароль.

Параметр **psw** набуває значень: будь-які символи і цифри довжиною до 8 байт.

Приклад:

```
AT+PASSW 876954321
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Network Password Ok: 876954321
```

******** Network Service Section**

Команда «AT+HI»

Призначення: знайти вузол, з яким встановлено з'єднання через Bluetooth (BLE).

Світлодіод "State" шуканого вузла блиматиме протягом двох секунд.

Приклад:

```
AT+HI
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> The node you are connected is flashing
```

Команда «AT+WHERE node»

Призначення: знайти вузол із зазначеною адресою.

Параметр **node** приймає значення: 1-254/00.01-13.15

Світлодіод "State" шуканого вузла блиматиме протягом двох секунд.

Приклад:

```
AT+WHERE 205 или AT+WHERE 12.14
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> The node you are looking for is flashing
```

Команда «AT+ERROR.log»

Призначення: показати журнал із помилками мережі та статистикою.

Приклад:

```
AT+ERROR.log 01.00
```

У вікні термінала відобразиться вміст журналу помилок кластера 1. Якщо параметр node не вказувати, то вузол "Net Master" відобразить вміст журналу помилок усієї мережі.

```
===== Network Channel Error =====
> SENDER  TARG    CNL     ERROR   QUALITY
  1.0     1.1     62      12      ***
  1.1     1.0     62      9       ****
  1.0     1.5     62      14      **
  1.5     1.0     63      16      **
```

Із записів журналу випливає, що канали 62 і 63 мають помилки.

При досягненні порога помилок у 20% мережа автоматично виключає канал зі списку доступних.

Команда «AT+NOD.scan»

Призначення: сканувати вузли мережі.

Приклад:

```
AT+NOD.scan
```

У вікні термінала відобразиться список знайдених вузлів мережі:

```
> Scan nodes begin. Wait...
*> Nodes found: 14
 20  21  22  23  24  25  26  27  28  29
123 204 207 208
```

Команда «AT+NOD.view»

Призначення: показати всі вузли в мережі.

Приклад:

```
AT+NOD.view
```

У вікні термінала відобразиться список знайдених вузлів мережі:

```
*> Nodes found: 15
 20  21  22  23  24  25  26  27  28  29
123 204 205 207 208
```

**** GPS Module Control Section

Команда «AT+GPS+»

Призначення: увімкнути функцію роботи з GPS модулем.

Приклад:

```
AT+GPS+
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> GPS state: ON
```

Команда «AT+GPS-»

Призначення: вимкнути функцію роботи з GPS модулем.

Приклад:

```
AT+GPS-
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> GPS state: OFF
```

Команда «AT+GPS.fix»

Призначення: зафіксувати поточні координати GPS модуля.

Приклад:

```
AT+GPS.fix
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> GPS coordinates fixed: зафіксовані координати широти і довготи
```

Якщо зафіксувати поточні координати GPS не вдалося, то з'явиться повідомлення:

```
> GPS fix error
> GPS Coordinates NOT fixed
```

Команда «AT+GPS.view»

Призначення: показати поточні координати GPS модуля.

Приклад:

```
AT+GPS.view
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> GPS current coordinates: поточні координати широти та довготи
```

****** BLE Module Control Section**

Команда «AT+BLE+»

Призначення: увімкнути функцію роботи з Bluetooth (BLE) модулем.

Приклад:

```
AT+BLE+
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Bluetooth state: ON
```

Команда «AT+BLE-»

Призначення: вимкнути функцію роботи з Bluetooth (BLE) модулем.

Приклад:

```
AT+BLE-
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> Bluetooth state: OFF
```

Команда «AT+BLE.config»

Призначення: подивитися поточні установки BLE модуля вузла.

Приклад:

```
AT+BLE.config
```

У вікні терміналу будуть відображені поточні установки BLE модуля:

```
*** BLE DEVICE CURRENT CONFIG ***
    BLE device Password: 12345678
    BLE device Name:
    111.203 Node For Test
```

Команда «AT+BLE.name n»

Призначення: присвоїти ім'я BLE модулю вузла.

Параметр n приймає значення: будь-які символи і цифри довжиною до 24 байт.

Приклад:

```
AT+BLE.name Node For Test
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> BLE Node name: Ok  111.203 Node For Test
```

Команда «AT+BLE.psw p»

Призначення: встановити пароль доступу до BLE модуля вузла.

Параметр **p** приймає значення: будь-які символи і цифри довжиною до 8 байт.

Приклад:

```
AT+BLE.psw 12345678
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> BLE password: Ok  12345678
```

******** Test Mode Section**

Команда «AT+STOP»

Призначення: зупинити поточний тест.

Приклад:

```
AT+STOP
```

У вікні термінала відобразиться повідомлення про те, що AT-команду виконано успішно.

```
> STOP
```

Поточний тест буде зупинено.

Команда «AT+COUNTER target[,period]»

Призначення: запустити тест для перевірки передавання даних між двома вузлами.

У тесті вузлом-відправником надсилається поточне значення лічильника вузлу-одержувачу.

Якщо вузол-одержувач отримав значення лічильника без помилок, то вузлу-відправнику надсилається квитанція ACK і вузол-відправник інкрементує значення лічильника.

Вузол-приймач фіксує кількість помилок лічильника і виводить на екран терміналу поточні значення.

Параметр **target** - адреса вузла-одержувача.

Параметр **period** - період надсилання тестових пакетів: 50...10000 ms. За замовчуванням: 500 ms.

Приклад:

```
AT+COUNTER 203,100
```

У вікні термінала вузла-відправника з'явиться повідомлення про те, що AT-команду виконано успішно, відправлене поточне значення лічильника і значення лічильника помилок, передане вузлом-одержувачем.

Якщо до вузла під'єднано та активовано модуль GPS, то відстань між вузлами також відображатиметься з похибкою 2 метри:

```
> Counter test started
>to:203|cnt:0000001|err:0|s=623 м|
>to:203|cnt:0000002|err:0|s=623 м|
>to:203|cnt:0000003|err:0|s=623 м|
>to:203|cnt:0000004|err:0|s=623 м|
```

У вікні термінала вузла-одержувача виводитиметься отримане значення лічильника:

```
>from:204|cnt:0000001|err:0|
>from:204|cnt:0000002|err:0|
>from:204|cnt:0000003|err:0|
>from:204|cnt:0000004|err:0|
```

Команда «AT+HELLO target[,period]»

Призначення: запустити тест для перевірки передачі даних між двома вузлами.

У тесті вузлом-відправником надсилається повідомлення "Hello from node".

Якщо вузол-одержувач отримав повідомлення, то у вікні терміналу вузла-одержувача буде відображатися повідомлення, адреса вузла-відправника і номер каналу, по якому було прийнято повідомлення.

Параметр **target** - адреса вузла-одержувача.

Параметр **period** - період надсилання тестових пакетів: 50...10000 ms. За замовчуванням: 500 ms.

Приклад:

```
AT+ HELLO 203,300
```

У вікні термінала вузла-одержувача з адресою 204 відобразиться повідомлення про те, що АТ-команду виконано успішно:

```
> Hello message start
0 203 Ok
0 203 Ok
0 203 Ok
```

У вікні термінала вузла-одержувача з адресою 203 виводитиметься повідомлення:

```
> Hello from node: 204 channel: 61
> Hello from node: 204 channel: 59
> Hello from node: 204 channel: 58
```

Команда «AT+HELLO.all [period]»

Призначення: запустити тест для перевірки передавання даних між кількома вузлами.

У тесті вузлом-відправником усім вузлам мережі по черзі надсилається повідомлення "Hello from node".

Якщо вузол-одержувач отримав повідомлення, то у вікні терміналу вузла-одержувача відображатиметься повідомлення, адреса вузла-відправника і номер каналу, яким було прийнято повідомлення.

Параметр **target** - адреса вузла-одержувача.

Параметр **period** - період надсилання тестових пакетів: 50...10000 ms. За замовчуванням: 500 ms.

Приклад:

```
AT+HELLO.all 100
```

У вікні термінала вузла-відправника відобразиться повідомлення про те, що АТ-команду виконано успішно:

```
> Hello to all message start
0 28 Ok
1 29 NO_CONNECTION
0 123 Ok
```

У вікні терміналу вузлів-одержувачів виводитиметься повідомлення:

```
> Hello from node: 203 channel: 76
> Hello from node: 203 channel: 80
> Hello from node: 203 channel: 79
```

Команда «AT+STRING target[,period][,length]»

Призначення: запустити тест для перевірки передавання даних між двома вузлами.

У тесті вузлом-відправником вузлу-одержувачу надсилається текстовий рядок довжиною, заданою користувачем.

Якщо вузол-одержувач отримав повідомлення, то у вікні терміналу вузла-одержувача відображатиметься прийнятий рядок.

Параметр **target** - адреса вузла-одержувача.

Параметр **period** - період надсилання тестових пакетів: 50...10000 ms. За замовчуванням: 500 ms.

Параметр **length** - довжина рядка: 1-150 байт. За замовчуванням: 50 байт.

Приклад:

```
AT+STRING 203
```

У вікні термінала вузла-відправника відобразиться повідомлення про те, що AT-команду виконано успішно:

```
> Test Sequence start
0 203 Ok
0 203 Ok
0 203 Ok
0 203 Ok
```

У вікні терміналу вузлів-одержувачів виводитиметься повідомлення:

```
123456789_123456789_123456789_123456789_123456789_
123456789_123456789_123456789_123456789_123456789_
123456789_123456789_123456789_123456789_123456789_
123456789_123456789_123456789_123456789_123456789_
```

Додаток 2

Формати AT-команд

Повний формат	**Короткий формат:**
AT+HELP	011
AT+NET.config	012
AT+NET.mode	013
AT+NOD.config	014
AT+NET.addr	015
AT+NOD.addr	015
AT+TRG.addr	017
AT+SRC.show	018
AT+SRC.hide	019
AT+CLU.config	021
AT+CLU.adm	022
AT+CLU.node	023
AT+CLU.targ	024
AT+ROUT.num	025
AT+CNL.main	026
AT+CNL.fixed	027
AT+CNL.multi	028
AT+CNL.scan	029
AT+CNL.view	02A
AT+CNL.set	02B
AT+POWER	031
AT+BITRATE	032
AT+BAUD	033
AT+UART.flow+	034
AT+UART.flow-	035
AT+START.hide	036
AT+START.show	037
AT+DEFAULT	038
AT+RESTART	039
AT+LOGIN	03A
AT+PASSW	03B
AT+HI	041
AT+WHERE	042
AT+ERROR.log	043
AT+NOD.scan	044
AT+NOD.view	045
AT+BLE+	046
AT+BLE-	047
AT+BLE.config	048
AT+BLE.name	049
AT+BLE.psw	04A
AT+STOP	051

AT+COUNTER 052
AT+HELLO 053
AT+HELLO.all 054
AT+STRING 055

AT+PIN.SET 061
AT+PIN.GET 062
AT+PIN.SET_CHANGE_NOTIFY 063
AT+PIN.CANCEL_CHANGE_NOTIFY 064
AT+PIN.SET_PERIOD_STATE_NOTIFY_MS 065
AT+PIN.SET_PERIOD_STATE_NOTIFY_SEC 066
AT+PIN.CANCEL_PERIOD_STATE_NOTIFY 067

AT+ADC.GET_ADC 071
AT+ADC.SET_CHANGE_NOTIFY 072
AT+ADC.CANCEL_CHANGE_NOTIFY 073
AT+ADC.SET_PERIOD_VALUE_NOTIFY_MS 074
AT+ADC.SET_PERIOD_VALUE_NOTIFY_SEC 075
AT+ADC.CANCEL_PERIOD_VALUE_NOTIFY 076

AT+DAC.SET 077

AT+PWM.SET 081
AT+PWM.DUTY 082
AT+PWM.CANCEL 083

AT+SERVO.SET 086
AT+SERVO.DEGREE 087
AT+SERVO.SET_SCAN 088
AT+SERVO.CANCEL_SCAN 089
AT+SERVO.CANCEL 08A

AT+GPS+ 091
AT+GPS- 092
AT+GPS.GET_COORDINATE 093
AT+GPS.GET_DATA 094
AT+GPS.FIX 095
AT+GPS.SET_CHANGE_NOTIFY 096
AT+GPS.CANCEL_CHANGE_NOTIFY 097
AT+GPS.SET_PERIOD_NOTIFY 098
AT+GPS.CANCEL_PERIOD_NOTIFY 099

Список літератури

1. Roberto Ierusalimschy "Programming in Lua". Lua.org,/August 2016./ISBN 8590379868 /388 p.
2. https://ESP8266.readthedocs.io/en/release/ (дата звернення: 12.03.2024).
3. https://coderlessons.com/tutorials/stsenarii/vyuchi-lua/lua-tutorial/(дата звернення: 12.03.2024).
4. https://ilovelua.narod.ru/about_lua.html/(дата звернення: 12.03.2024).
5. IEEE 802.15.4-2020 - IEEE Standard for Low-Rate Wireless Networks. Standards Committee : C/LM - LAN/MAN Standards Committee. 2020.05.06. URL: https://standards.ieee.org/standard/802_15_4-2020.html (дата звернення: 12.03.2024).
6. Recommendation G.9959. URL: https://www.itu.int/rec/T-REC-G.9959-201310-S!Amd1/en (дата звернення: 12.03.2024).
7. LoRaWAN™ Specification, N.Sornin (Semtech), M.Luis (Semtech), T.Eirich (IBM), T.Kramp (IBM), O.Hersent (Actility), V1.0, 2015 January.
8. Смирнов В.В., Смирнова Н.В. Бездротова локальна мережа класу Smart Home на базі модулів сплітерів-репітерів. Загальнодержавний міжвідомчий науково-технічний збірник Конструювання, виробництво та експлуатація сільськогосподарських машин. Кропивницький: ЦНТУ, 2021. Вип. 51. С. 195-202 (Фахове видання).
9. Смірнов В.В., Смірнова Н.В. Мобільна mesh-мережа для управління роєм об'єктів. Центральноукраїнський науковий вісник. Технічні науки. 2023. Вип. 7(38), ч.II. С. 3-11 (Фахове видання).
10. Смірнов В.В., Смірнова Н.В., Практична реалізація безпровідної мережі «Easy Net Everywhere». M 58 International security studios: managerial, technical, legal, environmental, informative and psychological aspects. International collective monograph. Volume I. NMBU, Research and Education. 2024. - 700 p. Oslo, Kingdom of Norway - 2024. C.665 - 697.

Навчальне електронне видання комбінованого використання.
Можна використовувати в локальному та мережному режимах

Смірнов Володимир Вікторович
кандидат технічних наук, доцент

Смірнова Наталія Володимирівна
кандидат технічних наук, доцент

Програмування пристроїв Internet Of Things на базі мікроконтролера ESP8266 мовою програмування LUA

Навчальний посібник

Редактор В.В. Смірнов
Технічний редактор В.В. Смірнов
Верстальник Н.В. Смірнова

Видавець
Центральноукраїнський національний технічний університет
25006, м. Кропивницький. пр. Університетський, 8,

тел. +38-091-608-75-02
E-mail: swckntu@gmail.com