NodeMCU-API 中文说明

Version 2.2.1 build 2019-08-19 By:dreamofTaotao

**概述**

1. 易编程无线节点/接入点。
2. 基于Lua5.1.4（没有debug&os模块）。
3. 装载异步事件驱动编程。
4. 超过65内置模块，但是该参考手册暂时只有17个常用内置模块，
5. 固件支持floating模式（integer仅支持小内存）。
6. 英文参考文献地址：[https://nodemcu.readthedocs.io](https://nodemcu.readthedocs.io/)。
7. 博主英文水平有限，欢迎指正错误。
8. 持续更新……

**ADC模块**

ADC模块提供了接入内置ADC。

在ESP8266中，仅有一个频道是电池电压有多路复用。依据设置在“esp init data”（107bit），其中一个也能用ADC去进行读取扩展电压，或者读取系统电压（VDD-3.3V），但是需要注意的是不能同时读取。

adc.force\_init\_mode()函数可以进行配置ADC的模式。注意在从一个系统到另一个系统需要重新开始（例如，电源重新连接、重置按钮或者是node.restart()），这个是必要的在更改生效前。

**函数表：**

|  |  |
| --- | --- |
| adc.force\_init\_mode() | 检查并在必要时重置在ESP数据初始化块中（init）中的ADC模式设置 |
| adc.read() | ADC读取采样 |
| adc.readvdd33() | 读取系统电压 |

**adc.force\_iniy\_mode()**

检查并在必要时重置在ESP数据初始化块中（init）中的ADC模式设置。

**使用语法：**

adc.force\_init\_mode(mode\_value)

**参数介绍：**

mode\_value：adc.INIT\_ADC或者adc.INIT\_VDD33其中之一。

**返回值：**

如果这个函数不得不改变当前模式时返回值为：true

如果当前模式已经配置完成时返回值为：flase

当返回值为：true时ESP需要需要重新启动使其生效，重启方式参考：电源重新连接、重置按钮或者是node.restart()。

**Example：**

-- 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))

**adc.read()**

ADC读取采样

**使用语法：**

adc.read(channel)

**参数介绍：**

channel：在ESP8266中总是0

**返回值：**

例子中的值（number）

如果ESP8266已经配置使用ADC读取系统电压，这个函数将总是返回65535。这个是硬件或者说是SDK的限制。

**Example：**

val=adc.read(0)

**adc.readvdd33()**

读取系统电压

**使用语法：**

adc.readvdd33()

**参数介绍：**

none

**返回值：**

系统电压，单位为毫伏（number）

如果ESP8266已经配置使用ADC实例扩展引脚进行采样，这个函数将总是返回65535。这个是硬件或者说是SDK的限制。

**cryto模块**

这个crypto模块为使用加密算法提供了多种函数。

接下来的加密解密算法模式被支持：

<1>“AES-ECB”：对于128bit的AES在ECB模式中（该种模式不推荐使用）

<2> “AES-ECB”：对于128bit的AES在CBC模式中。

下面hash（哈希）算法被支持：

<1>MD2（默认不可用，必须在app/include/user\_config.h文件中进行使能操作）

<2>MD5、SHA1、SHA256、SHA384、SHA512（在app/include /user\_config.h文件不使能）。

**函数表：**

|  |  |
| --- | --- |
| crypto.encrypt() | Lua字符串加密 |
| crypto.decrypt() | 解密之前加密的数据（字符串） |
| crypto.fhash() | 计算文件的加密哈希 |
| crypto.hash() | 计算一个Lua字符串的加密哈希 |
| crypto.new\_hash() | 创建可以添加任何数量字符串的哈希对象 |
| crypto.hmac() | 计算一个HMAC（哈希信息验证代码）的签名对于一个Lua字符串 |
| crypto.new\_hmac() | 创建可以添加任何数量字符串HMAC对象 |
| crypto.mask | 使用XOR（或非门）掩码应用于Lua字符串加密 |
| crypto.toBase64() | 提供二进制Lua字符串的Base64表示 |
| crypto.toHex() | 提供二进制Lua字符串的ASCII十六进制表示 |

**crypto.encrypt()**

Lua字符串加密。

**使用语法：**

crypto.encrypt(algo,key,plain[,iv])

**参数介绍：**

algo：将要在代码中支持使用的加密算法的名称。

key：加密的键设置为字符串；如果使用AES加密，那么这个字长必须设置成16 bytes。（解释：就是把需要加密的字符串变成字符串的样式）。

plain：需要加密的字符串；如果必要的话，将会自动的将0填充至为16-byte的边界。

iv：如果使用AES-CBC，初始化向量；如果没有赋值的话，默认为0。

**返回值：**

以二进制字符串形式加密的数据。对于AES返回值将总会是16字节长度的倍数。

**Example：**

print(crypto.toHex(crypto.encrypt(“AES-ECB”,”1234567890abcdef”,”Hi,I’m secret”)))

**crypto.decrypt()**

解密之前加密的数据。

**使用语法：**

crypto.decrypt(algo,key,cipher[,iv])

**参数介绍：**

algo：将要在代码中使用被支持的加密算法的名称。

key：加密的键设置为字符串；如果使用AES加密，那么这个字长必须设置成16 bytes。（解释：就是把需要加密的字符串变成字符串的样式）。

plain：要解密的密码文本（需要从crypto.encrypt()中获取） 。

iv：如果使用AES-CBC，初始化向量；如果没有赋值的话，默认为0。

**返回值：**

解密后的字符串.

注意解密后的字符串可能包括额外的0字节的填充在字符串结尾。一种剔除这个边缘值得方法是在解密字符串中加入：

“:match(“(.-)%z\*$”)”。另外还需要注意的是如果是在二进制下进行操作要小心，这个真实的字节长度可能需要被编码在数据中，并且“sub(1,n)”可以被用于去剔除填充值。

**Example：**

key = "1234567890abcdef"

cipher = crypto.encrypt("AES-ECB", key, "Hi, I'm secret!")

print(crypto.toHex(cipher))

print(crypto.decrypt("AES-ECB", key, cipher))

**crypto.fhash()**

计算文件的加密哈希

**使用语法：**

hash=crypto.fhash(algp,filename)

**参数介绍：**

algo：将要在代码中使用被支持的加密算法的名称，不区分大小写字符。

filename：需要哈希加密的文件路径。

**返回值：**

包括信息概要的二进制字符串。需要转化成文字版本（ASCII码十六进制字符）请参考使用函数：crypto.toHex()。

**Example：**

print(crypto.toHex(crypto.fhash(“SHA1”,”init.lua”)))

**crypto. hash()**

计算Lua字符串的加密哈希。

**使用语法：**

hash=crypto.hash(algo,str)

**参数介绍：**

algo：将要在代码中使用被支持的加密算法的名称，不区分大小写字符。

str：需要进行哈希算法的str字符串。

**返回值：**

包括信息概要的二进制字符串。需要转化成文字版本（ASCII码十六进制字符）请参考使用函数：crypto.toHex()。

**Example：**

print(crypto.toHex(crypto.hash(“SHA1”,”abc”)))

**crypto. new\_hash()**

创建一个能够添加任意数量字符串的哈希对象。对象有更新和完成的函数。

**使用语法：**

hashobj=crypto.new\_hash(algo)

**参数介绍：**

algo：将要在代码中使用被支持的加密算法的名称，不区分大小写字符。

**返回值：**

具有可用于更新和完成函数的UserData（用户数据）的对象。

**Example：**

hashobj = crypto.new\_hash("SHA1")

hashobj:update("FirstString")

hashobj:update("SecondString")

digest = hashobj:finalize()

print(crypto.toHex(digest))

**crypto. hmac()**

对一个Lua字符串计算一个HMAC（哈希信息验证代码）的签名。

**使用语法：**

signature = crypto .hmac(algo, str, key)

**参数介绍：**

algo：将要在代码中使用被支持的加密算法的名称，不区分大小写字符。

str：需要计算哈希的数据（字符串）。

key：用于签名的关键字（密钥），可能是二进制字符串。

**返回值：**

二进制字符串包含HMAC的签名。使用ctrypto.toHex()函数进行十六进制转换。

**Example：**

print(crypto.toHex(crypto.hmac(“SHA1”,”abc”,”mysecret”)))

**crypto.new\_hmac()**

创建一个可以添加任意数量字符串HMAC对象。对象具有更新和完成的函数。

**使用语法：**

hmacobj = crypto.new\_hmac(algo,key)

**参数介绍：**

algo：将要在代码中使用被支持的加密算法的名称，不区分大小写字符。

key：用于签名的关键字（密钥），可能是二进制字符串。

**返回值：**

具有可用于更新和完成函数的UserData（用户数据）的对象。

**Example：**

hashobj = crypto.new\_hmac("SHA1",”s3kr3t”)

hashobj:update("FirstString")

hashobj:update("SecondString")

digest = hashobj:finalize()

print(crypto.toHex(digesr))

**crypto.new\_hmac()**

使用XOR（或非门）掩码应用于Lua字符串加密.注意这不是一个适当的加密机制，然而有一些协议会使用他=它。

**使用语法：**

crypto.mask(message,mask)

**参数介绍：**

message：需要掩码的信息。

mask：应用于掩码，如果长度少于信息则进行重复。

**返回值：**

这个掩码信息是一个二进制字符串。可以使用crypto.toHex()获取（ASCII十六进制）文本格式。

**Example：**

print(crypto.toHex(crypto.mask(“some message to obscure”,”X0Y7”)))

**crypto.toBase64()**

提供一个二进制Lua字符串的Base64表示形式。

**使用语法：**

b64 = crypto.toBase64(binary)

**参数介绍：**

binary：输入字符串进行Base64编码。

**返回值：**

一个Base64数据形式的编码字符串。

**Example：**

print(crypto.toBase64(crypto.hash(“SHA1”,”abc”)))

**crypto.toBase64()**

提供一种二进制Lua字符串的ASCII十六进制表示形式。每一个被输入的字节被表示成两个十六进制的字符进行输出。

**使用语法：**

hexstr = crypto.toHex(binary)

**参数介绍：**

binary：输入字符串进行十六进制编码表示。

**返回值：**

一个ASCII十六进制字符串。

**Example：**

print(crypto.toHex(crypto.hash(“SHA1”,”abc”)))

**file模块**

file模块给文件系统和单个文件提供了接口。

file系统是平面文件系统，没有任何的文件分支（也就是文件夹的概念不存在）。

除了SPIFFS文件系统在内部的flash中，如果FatFS被使能允许的话，这个模块也可以在外置拓展的SD卡中接入FAT分区。例如，下面的代码：

*-- open file in flash:--打开在flash中文件*

if file.open("init.lua") t**hen**

print(file.read())

file.close()

**end**

*-- or with full pathspec—使用全路径打开相关文件*

file.open("/FLASH/init.lua")

*-- open file on SD card—打开SD卡中的文件*

if file.open("/SD0/somefile.txt") t**hen**

print(file.read())

file.close()

**end**

**函数表：**

|  |  |
| --- | --- |
| file.chdir() | 改变当前文件路径（和驱动器） |
| file.exists() | 确定指定的文件是否存在 |
| file.format() | 格式化文件系统 |
| file.fscfg() | 返回flash地址和文件系统的物理空间大小（字节） |
| file.fsinfo() | 返回文件系统的大小信息 |
| file.list() | 列出在文件系统中的所有文件 |
| file.mount() | 在SD卡中安装一个FatFs卷 |
| file.on() | 注册回调函数 |
| file.open() | 打开一个访问的文件，可能会创建它（用来写入模式） |
| file.remove() | 从文件系统中移除目标文件 |
| file.rename() | 文件重命名 |
| file.stat() | 获得表中目标文件或者目录属性 |
| Basic model | 在basic 模型中有最多只能有一个文件打开 |
| Object model | 文件被被文件创建的文件对象表示 |
| file.close()  file.obj:close() | 无条件关闭当前打开文件 |
| file.flush()  file.obj:close() | 刷新对文件系统的任何挂起写入操作，以防重新启动系统时丢失数据 |
| file.read()  file.obj:read() | 读取打开文件中的相关内容 |
| file.readline()  file.obj:readline() | 读取打开文件的下一行 |
| file.seek()  file.obj:seek() | 设置并且获取目标文件的位置（从文件开头开始测量），该位置由偏移量加上字符串从何处指定的基给出 |
| file.write()  file.obj:write() | 在打开文件中写字符串 |
| file.writeline()  file.obj:writeline() | 在打开文件中写字符串并在结尾处加”\n” |

**file.chdir()**

改变当前的文件目录和驱动。这被用在没有驱动或者目录。

在系统开始执行之后当前的目录默认由内部SPIFFS(Flash)的根目录。

注意：注意这个函数只有在烧录固件的时候有FatFS才可以使用。

**使用语法：**

file.chdir(dir)

**参数介绍：**

dir:文件的名称-/FLASH，/SD，/SD1等等。

**返回值：**

当成功时返回true，其他情况返回值为false。

**file.exists()**

确定指定的文件是否存在。

**使用语法：**

file.exists(filename)

**参数介绍：**

filename：需要被查看的文件名

**返回值：**

如果文件存在即使文件大小只有0字节，那么返回true；

当文件不存在时返回false。

**Example：**

files = file.list()

**i**f files["device.config"] then

print("Config file exists")

**end**

**if** file.exists("device.config") **then**

print("Config file exists")

**end**

**file.format()**

格式化文件系统。完全擦除之前的文件系统并且写一个新的。依靠在ESP中flash芯片的大小，这过程中可能会花费一些时间。

注意：这个函数不支持SD卡的格式化。

**使用语法：**

file.format()

**参数介绍：**

none（无）

**返回值：**

nil（无）

**file.fscfg()**

返回flash的地址和文件系统区的物理大小（字节大小）。

注意：该函数不支持SD卡。

**使用语法：**

file.fscfg()

**参数介绍：**

none（无）

**返回值：**

flash address：flash地址（数字类型）

size：大小 （数字类型）

**Example：**

print(string.format("0x%x", file.fscfg()))

**file.fsinfo()**

返回文件系统的大小信息。对于SPIFFS是字节型数据的，对于FatFS是kb型数据。

**使用语法：**

file.fsinfo()

**参数介绍：**

none（无）

**返回值：**

remaining：剩余大小 （数字类型）

used ：已使用空间（数字类型）

total：总空间大小 （数字类型）

**Example：**

*-- 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.fsinfo()**

列出文件系统中的所有文件。

**使用语法：**

file.list([pattern])

**参数介绍：**

none（无）

**返回值：**

Lua语法中的table（表格），如果没有模式被给的话，包含所有的{“文件名”:”文件大小”}。如果有模式被给，仅仅返回与该模式匹配的文件名（解释为传统的Lua模式，而不是Unix shell glob），file.list()将会抛出模式匹配期间遇到的错误。

**Example：**

**l = file.list();**

**for k,v in** pairs**(l) do**

print**(**"name:"**..k..**", size:"**..v)**

**end**

**file.mount()**

在SD卡中安装一个FatFs卷。

注意：该函数只在FatFS的相关文件烧录到固件中才会被支持，并且该函数不在内部flash中被支持。

**使用语法：**

file.mount(ldrv[, pin])

**参数介绍：**

ldrv：逻辑驱动器的名称，/SD0，/SD1等等。

pin：1~12，ss/cs的IO索引，如果省略，默认为8。

**返回值：**

卷对象

**Example：**

**vol = file.mount(**"/SD0"**)**

**vol:umount()**

**file.on()**

注册回调函数。

触发事件包括：

rtc：传递当前的日期和时间给文件系统。函数希望返回一个(table)表类型，其中包括当前日期和时间的year,mon,day,hour,min,sec。不支持内部flash。

**使用语法：**

file.on(event[, function()])

**参数介绍：**

event：字符串

function()：回调函数。如果省略的话将不会注册回调函数。

**返回值：**

nil

**Example：**

sntp.sync(server\_ip,

**function**()

print("sntp time sync ok")

file.on("rtc",

**function()**

**return** rtctime.epoch2cal(rtctime.get())

**end**)

**end**)

**file.open()**

打开一个访问的文件，可能会创建一个用来写模式。

当处理完文件的时候，必须使用file.close()关闭当前文件。

**使用语法：**

file.open(filename, mode)

**参数介绍：**

filename：将要被打开的文件的文件名。

mode：

“r”：读取模式（默认）

“w”：写模式

“a”：添加模式

“r+”：更新模式，所有之前的数据被保存

“w+”：更新模式，所有之前的数据被擦除

“a+”：添加更新模式，之前的数据被保存，但是只能在文件末尾进行写操作。

**返回值：**

如果文件被打开成功则返回文件对象。如果文件没有被打开或者不存在（读取模式）则返回nil。

**Example：（基础(basic)模式）**

*-- open 'init.lua', print the first line.*

**if** file.open("init.lua", "r") **then**

print(file.readline())

file.close()

**end**

**Example：（基础(object)模式）**

*-- open 'init.lua', print the first line.*

fd = file.open("init.lua", "r")

**if** fd **then**

print(fd:readline())

fd:close(); fd = **nil**

**end**

**file.remove()**

从文件系统中移除目标文件。但是，这个文件不能被打开。

**使用语法：**

file.remove(filename)

**参数介绍：**

filename：将会被移除的文件的文件名。

**返回值：**

nil

**Example：**

*-- remove "foo.lua" from file system.*

file.remove("foo.lua")

**file.rename()**

重命名文件。如果这个文件被打开，那么他将会先被关闭。

**使用语法：**

file.rename(oldname, newname)

**参数介绍：**

oldname：原文件名

newname：新文件名

**返回值：**

更改成功则返回true，更改失败则返回失败。

**Example：**

*-- rename file 'temp.lua' to 'init.lua'.*

file.rename("temp.lua","init.lua")

**file.stat()**

获取表中文件或目录属性。表的元素包括：

size：文件的大小（返回值为字节型）

name：文件名

time：表中时间戳信息。默认为1970-01-01 00:00:00，在SPIFFS中时间戳不被允许。year-mon-day-hour-min-sec

is\_dir：如果该项目是一个目录的话标志位为true，否则为false。

is\_rdonly：如果该项目是只读文件，那么标志位为true，否则为false。

is\_hidden：如果该项目被隐藏，那么标志位为true，否则为false。

is\_sys：如果该项目是系统属性，那么标志位为true，否则为false。

is\_arch：如果该项目为存档文件，那么标志位为true，否则为false。

**使用语法：**

file.stat(filename)

**参数介绍：**

filename：目标文件名

**返回值：**

包含文件属性的表

**Example：**

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**

**File access functions**

这个文件模块提供了多个函数在文件使用file.open()打开的文件后接入文件的内容。这些被用在了basic模块或者是object模块。

**Basic model：**

在basic模块中同时最多只可以打开一个文件。默认时，文件访问功能对该文件进行操作。如果另一个文件被打开，那么之前默认的文件需要在操作前被打开。

*-- open 'init.lua', print the first line.*

**if** file.open("init.lua", "r") **then**

print(file.readline())

file.close()

**end**

**Object model：**

文件被file.open()创建的文件对象表示。文件访问函数可用的作这个对象的方法，并且多个文件的对象能同时存在。

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**

特别注意：在一个应用中建议使用单个模块。如果同时使用两个模块的话将会出现不可预知的行为：从Basic模块中将会关闭默认文件的文件对象。从Object模块中关闭一个文件（如果两个是相同的文件也将关闭默认的文件）。

注意：在SPIFFS中打开文件的最大数量在编译时被

user\_config.h文件中SPIFFS\_MAX\_OPEN\_FILES所确定。

**file.close(),file.obj:close()**

关闭打开中的文件，强制执行。

**使用语法：**

file.close()

fd:close()

**参数介绍：**

none

**返回值：**

nil

**file.flush(),file.obj:flush()**

刷新任何被挂起的写入，确保再重启时没有数据丢失。用file.close()/fd:close()关闭正在打开的文件也会执行隐式刷新。

**使用语法：**

file.flush()

fd:flush()

**参数介绍：**

none（无）

**返回值：**

nil（无）

**Example：**

*-- 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**

**file.read(),file.obj:read()**

在正在打开中的文件中读取文本。

注意：这个函数在堆上临时分配2\*（请求的字节数），用于缓冲和处理读取的数据。默认块大小（文件读取块）为1024字节，被认为是安全的。将此值推高4倍或者更高可能会导致堆溢出，具体取决与应用程序。选择n\_or\_char参数时需要考虑这一点。

**使用语法：**

file.read([n\_or\_char])

fd:read([n\_or\_char])

**参数介绍：**

n\_or\_char：

<1>如果没有值被传入，那么读取到达FILE\_READ\_CHUNK（块字节）（bytes字节），或者全部的文件（以最小为准）。

<2>如果传入一个数字为n，那么将读取n个字节或者整个文件（不论有多小）。

<3>如果传入了一个字符串包括单独一个字符char，那么读取直到char出现在文件中，文件块字节已经被读，或者达到了EOF。

**返回值：**

文件的内容作为一个字符串，或者当文件为EOF是为nil。

**Example：（basic model）**

*-- print the first line of 'init.lua'*

**if** file.open("init.lua", "r") **then**

print(file.read('\n'))

file.close()

**end**

file.write('baz')

**Example：（object model）**

*-- print the first 5 bytes of 'init.lua'*

fd = file.open("init.lua", "r")

**if** fd **then**

print(fd:read(5))

fd:close(); fd = n**il**

**end**

**file.readline(),file.obj:readline()**

读取已打开文件的下一行。行被定义成0或者更多的字节在结尾的时候有EOF(‘\n’)字节。如果下一行超过了1024字节，这个函数进京返回第一个1024字节。

**使用语法：**

file.readline()

fd:readline()

**参数介绍：**

none（无）

**返回值：**

文件字符串形式的内容，一行一行的，包括EOF(‘\n’)->换行符。当遇到换行符EOF(‘\n’)时返回nil。

**Example：（object model）**

*-- print the first line of 'init.lua'*

**if** file.open("init.lua", "r") **then**

print(fd:readline())

file.close()

**end**

**file.seek(),file.obj:seek()**

设置并且获取文件的位置，从文件开头的位置开始测量，该位置由偏移量加上字符串指定的基得出。

**使用语法：**

file.seek([whence [, offset]])

fd:seek([whence [, offset]])

**参数介绍：**

whence：”set”:基位置是0（文件开始的地方）

“cur”:当前的位置，（默认值）

“end”:文件末尾的位置

offset：偏移量默认为0

如果没有给函数体内传入任何的参数，这个函数就会返回当前文件的偏移量。

**返回值：**

结果文件位置，或出错时为零

**Example：（basic model）**

**if** file.open("init.lua", "r") **then**

*-- seek the first 5 bytes of 'init.lua'*

file.seek(“set”,5)

print(file.readline())

file.close()

**end**

**file.write(),file.obj:write()**

给正在打开的文件写一个字符串。

**使用语法：**

file.write(string)

fd:write(string)

**参数介绍：**

string：给文件将要写的字符串。

**返回值：**

如果文件被写成功，则返回true，反之则返回值为0（非零即为真）。

**Example：（basic model）**

*-- 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**

**Example：（object model）**

*-- 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**

**file.writeline(),file.obj:writeline()**

给一个以打开的文件写一个字符串，并且在文件的末尾处添加换行符’\n’。

**使用语法：**

file.writeline(string)

fd:writeline(string)

**参数介绍：**

string：将要给文件写的字符串。

**返回值：**

如果被写成功则返回true，否则（错误）返回nil。

**Example：（basic model）**

*-- 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**

**GPIO模块**

这个模块提供了GPIO（通用输入、输出）接入系统。

所有的访问都是局域I/O口的索引值在NodeMCU开发工具包中，不是内部的GPIO引脚。例如，D0引脚在开发工具包中对应的是内部的GPIO第16引脚。

如果不是应用NodeMCU开发工具包，请参考下列的ESP8266gpio引脚相应的GPIO引脚索引值。

|  |  |  |  |
| --- | --- | --- | --- |
| IO index | ESP8266 pin | IO index | ESP8266 pin |
| 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)只能被用作GPIO的读写操作。不支持开漏/中断/PWM/i2C/OW。

**函数表：**

|  |  |
| --- | --- |
| gpio.mode() | 对GPIO模式进行初始化，设置引脚的输入输出方向，并且选择内部弱上拉 |
| gpio.read() | 读取GPIO引脚的数字值 |
| gpio.serout() | 根据延迟时间序列（以为微秒为单位）对输出进行序列化 |
| gpio.trig() | 对正在运行引脚的中断建立或者清除一个回调函数 |
| gpio.write() | 设置GPIO引脚的数字量 |
| gpio.pulse | 这一组API可以对多个引脚产生一个精确的定时的脉冲列 |
| gpio.pulse.build | GPIO的构建 |
| gpio.pulse:start | 开始输出操作 |
| gpio.pulse:getstate | 返回当前状态 |
| gpio.pulse:stop | 在未来某个时刻停止输出操作 |
| gpio.pulse:cancle | 立即停止输出操作 |
| gpio.pulse:adjust | 增加或者减少时间使得能够习惯于最小最大延时场景 |
| gpio.pulse:update | 在输出项目中更改特殊一步的内容 |

**gpio.mode()**

对GPIO的模式进行初始化，设置引脚的输入输出方向，并且选择内部弱上拉。

**使用语法：**

gpio.mode(pin, mode [, pullup])

【注意：括号内的内容可加可不加。】

**参数介绍：**

pin：根据IO索引查找相应设置的引脚

mode：gpio.OUTPUT,gpio.OPENDRAIN,gpio.INPUT,gpio.INT(中断模式)，4个其中有一个（上面四个分别是(按序)：输出、开漏、输入、中断）

pullup：gpio.PULLUP使能弱上拉电阻；默认是gpio.FLOAT(浮空)

**返回值：**

nil

**Example：**

gpio.mode(0,gpio.OUTPUT)

**gpio.read()**

读取GPIO引脚数字量。

**使用语法：**

gpio.read(pin)

**参数介绍：**

pin：根据IO索引表查找，需要读取的引脚

**返回值：**

一个数字类型，0表示低（low），1表示高（high）

**Example：**

-- read value of gpio 0.

gpio.read(0)

**gpio.serout()**

根据延迟时间序列（以为微秒为单位）对输出进行序列化。在每个延迟之后，每一个引脚都被切换。在上一次循环和上一次延时之后引脚不被切换。

这个函数有两种工作模式：\*同步是分辨率-50微秒，限制最大总持续时长值。所有的时间间隔，\*异步时钟-同步时钟操作，粒度较小，但实际上持续时间不受限制。

异步时钟是否被选择由当前的回调函数的参数。如果存在并且是函数类型，则函数将异步进行，并且在序列结束时调用回调函数。如果这个参数是数字，那么函数仍将继续异步但是当做完之后不会回调函数。

对于这个异步版本，最小的延时函数应该不小于50微秒并且最大延时时长为0x7fffff微秒（~8.3秒）。在这个模式中函数不会在输出序列完成前妨碍栈堆并且立即返回。HW定时器FRC1\_SOURCE模式被用于改变这个状态。当只有一个单独的硬件定时器时，将会哪些模块可以同时被使用有限制，如果这个定时器正在被使用中，那么将会有错误出现。

注意：同步变化（没有返回值或者返回值为nil）函数可以阻塞栈堆并且当任何模块想要使用的话需要参考SDK指南。如果不这样做的话可能会导致WiFi问题或者奔溃或者重启。简而言之，所有的延时和循环次数的总和不能超过15毫秒。

**使用语法：**

gpio.serout(pin, start\_level, delay\_times [, cycle\_num[, callback]])

**参数介绍：**

pin：根据IO索引图查找想要使用的引脚。

start\_level：开始的电平，gpio.HIGH或者gpio.LOW两者之一。

delay\_times：微秒级的延时时长数组， 数组中的每一项对应切换的引脚。

cycle\_num：在序列中运行的可选次数（默认为1）.

callback：可选择的回调函数或者是数字，如果存在返回当前的函数并且执行异步操作。

**返回值：**

nil

**Example：**

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()**

对一个引脚建立或者清除一个回调函数执行中断。（有中断触发方式）

如果GPIO\_INTERRUPT\_ENABLE在编译的的时候没有被定义，那么这个函数不可用。

**使用语法：**

gpio.trig(pin, [type [, callback\_function]])

**参数介绍：**

pin：1-12，引脚触发，可在IO中索引。注意：第0号引脚不支持中断。

type：”up”,”down”,”both”,”low”,”high”，分别表示上升沿、下降沿、双边沿、低电平和高电平触发方式。如果这个类型是”none”或者是省略的话，那么回调函数被移除并且不会使能中断。

callback\_function(level,when,eventcount)：触发时的回调函数。在这里的level表示的是在中断作为第一个参数对于回调函数。事件的时间戳作为第二个参数进行传递。这是一个毫秒级并且同样基于tmr.now()。这个时间戳在中断中被抓取的，并且比在回调函数中或取的一致。这个时间戳通常是检测到第一个中断，但是过载条件下可能是一个更晚的中断。这个事件次数是为此种中断回调的次数。这个最好在边沿触发中并且使能边沿的次数。然而，当心开启反弹，你可能得到多个中脉冲在一次开关闭合中。计数工作最好是边沿由数字产生。

如果这个函数是默认的话，那么之前的函数将会被使用。

**返回值：**

nil

**Example：**

**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**

**gpio.write()**

给GPIO引脚写数字量。

**使用语法：**

gpio.write(pin, level)

**参数介绍：**

pin：根据IO索引查找相应的引脚编号

level：gpio.HIGH或者gpio.LOW二选一。

**返回值：**

nil

**Example：**

*-- 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)

**gpio.pulse**

这包括了一组允许给多个引脚产生准确的时间脉冲列的API。这个类似于serout的API，但是这个API能处理多个引脚并且可以更好地控制时间。

这个基本的想法是建立一个gpio.pulse对象，并且用对象的方式控制它。某一时刻仅能激活一个gpio.pulse对象。这个对象被建立从一个表的数组中，每个内部的表代表一个激活操作并且在下一个激活操作之前需要花一些时间（需要延时）。

这其中的一个是用于去产生双极脉冲驱动时钟移动，你想在偶数秒是第一引脚有脉冲，并且奇数秒给第二引脚有脉冲。:getstate和:adjust可以用于保持同步脉冲给RTC时钟（它本身与NTP同步）。

注意：这个附属的模块默认是不被使能的（默认关闭）。如果需要使用他，那么在烧录固件之前在app/include/user\_modules.h文件中取消LUA\_USE\_MODULES\_GPIO\_PULSE该语句的注释，这样则使能该模块。

使用该功能，决定于你希望产生重复多少次数的脉冲列。取决于你将会使用的GPIO引脚的数量。之后绘制你想要去执行的相应顺序的一个表格。之后你可以建立一个传递至gpio.pulse.build的表的结构。例如，对于两个异相的方波，你可以：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Step | Pin 1 | Pin 2 | Duration(us) | Next Step |
| 1 | High | Low | 100,000 | 2 |
| 2 | Low | High | 100,000 | 1 |

这将会（当建立并且开始的时候）仅仅去执行Step1（按规定设置输出引脚），并且之后100,000us，这个将会执行Step2。这个交换的输出引脚并且在下一次转化至Step1之前等待100,000，这个对5Hz的两个反相方波是有效的输出对Pin1和Pin2。这个频率将会比5Hz稍微低一些，因为它是由软件产生并且中断屏蔽能够延迟执行下一步。为了最好的接近5Hz，你可以允许每次执行下一步的时间间隔有微弱的变化。以至于会调整每一步的时间间隔，总体上输出是5Hz。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Step | Pin1 | Pin2 | Duration(us) | Range | Next Step |
| 1 | High | Low | 100,000 | 90,000-110,000 | 2 |
| 2 | Low | High | 100,000 | 90,000-110,000 | 1 |

当转至这个表结构下面见会描述，当下一步的编号多余当前步骤时，你不需要去指定的任何特殊的内容。当指定一个输出的顺序步骤，你必须指定这个需要多长时间执行一次。这个迭代次数能达到4,000,000,000(事实上适合任何一个无符号的32位的整数类型)。如果这不足够重复，那么循环可以被嵌套，就像下面的例子一样：

{

{ [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 }

}

这个循环的次数在Step2将会造成1,000,000,000次脉冲被输出（在1KHz）。这将会持续11天。在这一点，他将继续执行1KHz11天的触发器在Step3。这个过程将重复1,000,000,000次（这大概有3000万年）。

**gpio.pulse**

它建立了gpio.pulse对象来自所提供的参数。（正如下面表所描述的）。

**使用语法：**

gpio.pulse.build(table)

**参数介绍：**

table：这是作为一个说明的数组，每个说明被一个像下面的表所代表：

<1>所有的键被认为是引脚的索引值，每一个值是设置在GPIO行的对应值。例如：{[1]=gpio.HIGH}将会设置第1号引脚设置为高电平。注意这个是NodeMCU的引脚号，不是ESP8266的GPIO的引脚值。锁哥引脚可以被同时设置。注意任何有效的GPIO引脚可以被使用，包括Pin0。

<2>delay：特指在移动到下一个状态的设置的毫秒级的引脚值.这个真是的延时或许比它依赖于是否中断在最后被使能更加的长。这个最大的值是64,000,000-例如：1bit>1min。

<3>min和max：能够被指定（伴随着延迟）。当一个count和loop是完整的时候，下一个状态实在loop中的（前提是计数器不能是0），这个计数器接口作为一个32位无符号整数。例如，它的范围可以达到4,000,000,000。这第一个状态是状态1。这个loop很像是goto指令，因为可以跳转至下一条指令。

**返回值：**

gpio.pulse对象

**Example：**

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**)

这将会产生一个方波在Pin1和Pin2之间，但是他们将完全反相。在10秒钟之后，这方波将会伴随着Pin2引脚至高而结束。

注意：你必须在开始输出频率之前设置输出模式（gpio.OUTPUT或者是GPIOOPENDRAIN两者其中之一），否则什么都不会发生。

**gpio.pulse:start**

开始输出操作。

**使用语法：**

pulser:start([adjust, ] callback)

**参数介绍：**

adjust：这个是一个毫秒级的数据在添加至下一次调整之前。如果这个值是非常大以至于他将会超过min和max的值，那么剩余的部分将会保持到下一次调整。

callback：这个回调当这个脉冲完成的时候被执行。这个回调在这四个相同的参数被作为返回值时进行调用gpio.pulse:getstate。

**返回值：**

nil

**Example：**

pulser:start(**function**(pos, steps, offset, now)

print (pos, steps, offset, now)

**end**)

**gpio.pulse:getstate**

返回当前的状态。这四个值也被传入回调函数。

**使用语法：**

pulser:getstate()

**返回值：**

position：这个当前激活状态的索引值。这第一个状态是1。如果输出操作被完成那么返回值为nil。

steps：是一个状态的数字。已经被执行（包括现在的1）。当有循环的时候允许进程的管理。

offset：是一个毫秒的时刻直到下一个状态被传入。一旦输出操作完成了这个将会被消除。

now：是一个tmr.now()函数返回当前时刻的值，当偏移被计算。

**Example：**

pos, steps, offset, now = pulser:getstate()

print (pos, steps, offset, now)

**gpio.pulse:stop**

在未来某个时刻停止输出操作。

**使用语法：**

pulser:stop([position ,] callback)

**参数介绍：**

position：停止时刻的引脚索引值。这将会在进入这个状态时停止。如果不指定的话，将在下一次状态传入时停止。

callback：被调用（伴随着同样的参数被返回:getstate）当这个操作已经被停止。

**返回值：**

如果这个停止将要发生时返回true。

**Example：**

pulser:stop(**function**(pos, steps, offset, now)

print (pos, steps, offset, now)

**end**)

**gpio.pulse:cancel**

将会立即执行输出停止操作。

**使用语法：**

pulser:cancel()

**返回值：**

position：停止时刻的引脚索引值。这将会在进入这个状态时停止。如果不指定的话，将在下一次状态传入时停止。

steps：是一个状态的数字。已经被执行（包括现在的1）。当有循环的时候允许进程的管理。

offset：是一个毫秒的时刻直到下一个状态被传入。一旦输出操作完成了这个将会被消除。

now：是一个tmr.now()函数返回当前时刻的值，当偏移被计算。

**Example：**

pulser:cancel(**function**(pos, steps, offset, now)

print (pos, steps, offset, now)

**end**)

**gpio.pulse:adjust**

这将会添加（或者减少）时间将会习惯min/max延时时间。这个也是有用的如果你正在尝试对一个特殊的时间或者额外的时间同步一个特殊的状态。

**使用语法：**

pulser:adjust(offset)

**参数介绍：**

offset：是一个毫秒级的数字被用于后来的min/max延时。这将覆盖被挂起的偏移量。

**返回值：**

position：停止时刻的引脚索引值。这将会在进入这个状态时停止。如果不指定的话，将在下一次状态传入时停止。

steps：是一个状态的数字。已经被执行（包括现在的1）。当有循环的时候允许进程的管理。

offset：是一个毫秒的时刻直到下一个状态被传入。一旦输出操作完成了这个将会被消除。

now：是一个tmr.now()函数返回当前时刻的值，当偏移被计算。

**Example：**

pulser:adjust(177)

**gpio.pulse:update**

这个能够改变在输出项目中特殊的一步的内容。真能被用于调整延时时间，或者甚至是引脚。这个不能被用于去移除全部或者添加新的条项或者是添加新的条项。改变这个计数器将会改变初始值，但是不能改变当前的递减值。

**使用语法：**

pulser:update(entrynum, entrytable)

**参数介绍：**

entrynum：是初始脉冲序列定义的一个条目编号。第一个编号的数值是1。

entrytable：是一个包含与gpio.pulse.build同样键的表。

**返回值：**

nothing（无）

**Example：**

pulser:update(1, { delay=1000 })

**HTTP模块**

基于HTTP客户机模块，这个模块提供了例如GET/POST/PUT/DELETE接口在HTTP(s)的基础上，当然也可以自定义请求。由于ESP8266内存的限制，页面/内容的大小都被可用内存所限制。如果尝试接收大的网页将会导致报错。如果必须接入大的网页/内容，请考虑在数据中使用net.createConncteion()和stream。

注意：使用这个模块不可以多个同时执行HTTP请求。

每一个请求方法需要一个回调，当从服务器收到响应时调用。第一个参数是状态码，也是常规的HTTP状态码，或者是-1表示一个DNS，连接或者内存溢出失败，或者连接超时（一般是60秒）。

对于每个操作都可能提供一个自定义HTTP头部或者是覆盖标准的头部。默认的Host头是从URL中推断出来的，用户代理是ESP8266。注意，不能覆盖连接头！他总是设置为关闭。

HTTP重定向（HTTP状态值300-308）被自动跟踪到20个限定，以避免可怕的重定向循环。

当回调函数被调用，将会被传入一个HTTP状态代码，这个内容作为它收到的值，并且一个回应头部的表。为了方便接入，所有的头部都变得更小了。如果有多个头部重名，那么仅仅最后一个被返回。

**SSL/TLS支持：**

注意net模块中有相关的定义。

|  |  |
| --- | --- |
| http.delete() | 执行HTTP删除请求 |
| http.get() | 执行HTTP GET请求 |
| http.post() | 执行HTTP POST请求 |
| http.put() | 执行HTTP PUT请求 |
| http.request() | 对于任何的HTTP方法执行自定义HTTP请求 |

**http.delete()**

执行HTTP删除请求。注意不支持同时有多个请求。

**使用语法：**

http.delete(url, headers, body, callback)

**参数介绍：**

url：要获取的URL，包括http://或者https://前缀的。

hearders：选择要添加的可选附加头，包括\r\n；可以为nil。

body：需要发送的内容；必须已经被编码成适当的格式，但是可以为空。

callback：当有错误发生或者是接收到响应时被调用的回调函数；这三个参数status\_code,body,headers随着被调用。当出现错误时status\_code被设置成-1。

**返回值：**

nil

**Example：**

http.delete('http://httpbin.org/delete',

"",

"",

**function**(code, data)

**if** (code < 0) **then**

print("HTTP request failed")

**else**

print(code, data)

**end**

**end**)

**http.delete()**

执行一个HTTP GET请求。注意该函数不支持同时多个请求。

**使用语法：**

http.get(url, headers, callback)

**参数介绍：**

url：要获取的URL，包括http://或者https://前缀的。

hearders：选择要添加的可选附加头，包括\r\n；可以为nil。

callback：当有错误发生或者是接收到响应时被调用的回调函数；这三个参数status\_code,body,headers随着被调用。当出现错误时status\_code被设置成-1。

**返回值：**

nil

**Example：**

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()**

执行HTTP POSTq请求。注意该函数不支持同时多个请求。

**使用语法：**

http.post(url, headers, body, callback)

**参数介绍：**

url：要获取的URL，包括http://或者https://前缀的。

hearders：选择要添加的可选附加头，包括\r\n；可以为nil。

callback：当有错误发生或者是接收到响应时被调用的回调函数；这三个参数status\_code,body,headers随着被调用。当出现错误时status\_code被设置成-1。

**返回值：**

nil

**Example：**

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()**

执行HTTP PUT请求。注意该函数不支持同时多个请求。

**使用语法：**

http.put(url, headers, body, callback)

**参数介绍：**

url：要获取的URL，包括http://或者https://前缀的。

hearders：选择要添加的可选附加头，包括\r\n；可以为nil。

callback：当有错误发生或者是接收到响应时被调用的回调函数；这三个参数status\_code,body,headers随着被调用。当出现错误时status\_code被设置成-1。

**返回值：**

nil

**Example：**

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()**

对于任何的HTTP方法，执行一个自定义的HTTP请求。注意该函数不支持同时多个请求。

**使用语法：**

http.request(url, method, headers, body, callback)

**参数介绍：**

url：要获取的URL，包括http://或者https://前缀的。

method：要使用的HTTP方法例如：”GET”,”HEAD”,”OPTIONS”等等。

hearders：选择要添加的可选附加头，包括\r\n；可以为nil。

callback：当有错误发生或者是接收到响应时被调用的回调函数；这三个参数status\_code,body,headers随着被调用。当出现错误时status\_code被设置成-1。

**返回值：**

nil

**Example：**

http.request("http://httpbin.org", "HEAD", "", "",

function(code, data)

if (code < 0) then

print("HTTP request failed")

else

print(code, data)

end

end)

**I²C模块**

|  |  |
| --- | --- |
| i2c.address() | 设置下一次传输时i2c地址和的读取/写入模式 |
| i2c.read() | 读取数据对于可变字节数 |
| i2c.setup() | 初始化I2c模块 |
| i2c.start() | 发送一个I2C开始的条件 |
| i2c.stop() | 发送一个I2C停止的条件 |
| i2c.write() | 在I2C总线中写入数据 |

**i2c.address()**

设置下一次传输时的I2C的地址和读取/写入的模式。

**使用语法：**

i2c.address(id, device\_addr, direction)

**参数介绍：**

id：总是0。

device\_addr：1位的设备地址，切记在I2C设备中\_addr表示高7位，后跟一个方向位。

direction：对于写入模式的话，该参数为i2c.TRANSMITTER；对于读取模式的话，该参数为i2c.RECEIVER。

**返回值：**

如果接受到ACK的话返回值为true，反之则为false。

**i2c.read()**

对于可变字节数读取数据。

**使用语法：**

i2c.read(id, len)

**参数介绍：**

id：总是0。

len：数据字节数。

**返回值：**

一个接受到的string（字符串）。

**Example：**

id = 0

sda = 1

scl = 2

*-- initialize i2c, set pin1 as sda, set pin2 as scl*

i2c.setup(id, sda, scl, i2c.SLOW)

*-- user defined function: read from reg\_addr content of dev\_addr*

**function** **read\_reg**(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(0x77, 0xAA)

print(string.byte(reg))

**i2c.setup()**

初始化I2C模块。

**使用语法：**

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

**参数介绍：**

id：总是0。

pinSDA：编号1~12，参照IO表。//串行时钟信号线

pinSCL：编号1~12，参照IO表。//串行时钟信号线

speed：只有i2c.SLOW被支持。

**返回值：**

speed：被选择的speed。

**i2c.start()**

发送I2C启动的条件。

**使用语法：**

i2c.start(id)

**参数介绍：**

id：总是0。

**返回值：**

nil

**i2c.stop()**

发送I2C 停止的条件。

**使用语法：**

i2c.stop(id)

**参数介绍：**

id：总是0。

**返回值：**

nil

**i2c.write()**

向I2C总线中写入数据。数据项可以是多种数字，字符串或者是Lua表。

**使用语法：**

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

**参数介绍：**

id：总是0。

data：数据项可以是多种数字，字符串或者是Lua表。

**返回值：**

number：写入数据的字节量。

**Example：**

i2c.write(0, "hello", "world")

**MQTT模块**

提前解释一下：下面的QoS参数是服务质量，指的是一个网络能够利用的各种基础技术，为指定的网络通信提供更好的服务能力，是网络的一种安全机制，用来解决网络延迟和阻塞等问题的技术。

这个客户端依赖于MQTT的3.1.1版本。你需要确定好代理支持透传并且正确的配置了Ver3.1.1。客户机与运行MQTT3.1的代理程序前后不兼容。

|  |  |
| --- | --- |
| mqtt.Client() | 创建一个MQTT客户端 |
| mqtt.client:close() | 关闭跟代理的连接 |
| mqtt.client:connect() | 连接到被给”host,port,secure options”指定的代理 |
| mqtt.client:lwt() | 设置最终的遗嘱（最后的信息）选项 |
| mqtt.client:on() | 给一个事件注册一个回调函数 |
| mqtt.client:publish() | 发布一条消息 |
| mqtt.client:subscribe() | 订阅一个或者多个主题 |
| mqtt.client:unsubscribe() | 对一个或者多个主题取消订阅 |

**mqtt.write()**

创建一个MQTT客户机。

**使用语法：**

mqtt.Client(clientid, keepalive[, username, password, cleansession, max\_message\_length])

**参数介绍：**

clientid:客户端ID。

keepalive：保活时间。

username：用户名。

password：用户密码。

cleansession：0/1对应false、true。默认是true（1）。

max\_message\_length：接收最大信息的长度。默认是1024。

**返回值：**

MQTT客户端。

**注意：**

根据MQTT指定了最大发布信息的长度是256MB。对于NodeMCU的处理能力，这个数据量恐怕过大。为了避免内存溢出，在NodeMCU中设置了最大接收信息的长度。这个被参数：max\_message\_length控制。在实际应用中，这也只影响传入的发布信息，因为所有常规的操作都非常的小。默认情况下1024倍选定作为在NodeMCU2.1.1之前的版本中的限制。

注意最大的信息长度指的是整个MQTT的信息长度，包括固定和变量头、包ID（如果适用的话）和有效负载。有关详细信息，请参考MQTT规范。

如果定义过，那么所有信息将会比max\_message\_length更长（特别是）传输给过流的回调。这剩余的信息将会被丢掉。并且所有的后续消息都应按照预期处理。如果请求Q0S级别1或者2，即使应用程序栈堆无法处理求其的小事，这些消息仍被确定。

对堆栈内存将被用于缓冲所有的信息，跨越多个TCP数据包。对全部的信息有一个单独的分配将被展示当数据头被看到的时候，这样为了避免堆碎片化。如果分配失败，那么将会断开MQTT的对话连接。通常大于max\_message\_length的消息将不会被存储。

注意堆分配可能发生即使私有的消息没有配置的消息大。例如，代理可以快速连续的发送多个小的消息，这也可能进入同一个TCP包中。如果在数据包中最后的消息没有满，那么堆栈将会在等待下一次TCP包的时候保存不完整的消息。

对于一个消息而言典型的单TCP包的最大大小是1460字节，但是这依靠于网络的MTU配置，任何包碎片和上面被描述的多个信息在同一个TCP包中。

**Example：**

*-- 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("connect", **function**(client) print ("connected") **end**)

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, 0, **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("failed reason: " .. reason)

**end**)

m:close();

*-- you can call m:connect again*

**MQTT Client**

**mqtt.client:close()**

关闭与代理的连接。

**使用语法：**

mqtt:close()

**参数介绍：**

none

**返回值：**

如果成功则返回true，反之返回false。

**mqtt.client:connect()**

连接至有主机、端口、安全选项的指定的代理。

**使用语法：**

mqtt:connect(host[, port[, secure[, autoreconnect]]][, function(client)[, function(client, reason)]])

**参数介绍：**

host：主机、域名或者IP地址。

port：代理的端口号默认为1883。

secure：0/1对应false/true，默认为0，注意NET模块中的相关约束。

autoreconnect：0/1对应false/true，默认为0。这个选择已弃用。

function(client)：当建立连接时的回调函数。

function(client,reason)：当连接没有建立的时候的回调函数。不应该再调用回调函数。

**返回值：**

如果成功则返回true，反之返回false。

**注意：**

不要使用autoreconnect。重复一遍，不要使用autoreconnect。你应该处理这个暴露出来的错误并且应用于你的APP。特别的，上面的cleansession的默认值是true，因此当连接因任何原因丢失，所有的订阅全部被销毁。

为了实现一个自动连接，并在错误的回调中处理相应的错误。请参考：

**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**

事实上，连接的函数应该做一些有用的事情。

下面解释了自动重连的函数如何工作或者不工作。

当自动重连被设置，那当他被打断是将会重新建立。当然在重新连接是任何错误不会暴露，并且如果cleansession设置是true时，那么订阅全部丢失。然而，如果第一次连接失败，那么不会重新连接，并且会通过回调返回错误的值。如果第一次的连接客户端连接到了服务器并且获得了回应MQTT连接请求成功的包，那么被认为成功。这也就推测出用户名密码都正确。

**Connection failure callback reason codes：（连接失败是返回代码）：**

|  |  |  |
| --- | --- | --- |
| 常量 | 值 | 描述 |
| mqtt.CONN\_FAIL\_SERVER\_NOT\_FOUND | -5 | 没有代理在指定的IP地址上侦听 |
| mqtt.CONN\_FAIL\_NOT\_A\_CONNACK\_MSG | -4 | 根据协议的要求，代理人不是一个CONNACK |
| mqtt.CONN\_FAIL\_DNS | -3 | DNS查看失败 |
| mqtt.CONN\_FAIL\_TIMEOUT\_RECEIVING | -2 | 来自代理的CONNACK连接超时 |
| mqtt.CONN\_FAIL\_TIOME\_SENDING | -1 | 尝试发送一个连接信息超时 |
| mqtt.CONNACK\_ACCEPTED | 0 | 没有错误，注意：这将不会触发失败回调 |
| mqtt. CONNACK\_REFUSED\_PROTOCOL\_VER | 1 | 当前代理的版本不是3.1.1 |
| mqtt. CONNACK\_REFUSED\_ID\_REJECTED | 2 | 指定的客户端ID被代理拒绝（参照mqtt.Client()） |
| mqtt.CONNACK\_REFUSED\_SERVER\_UNAVAILABLE | 3 | 服务器不允许接入 |
| mqtt.CONNACK\_REFUSED\_BAD\_USER\_OR\_PASS | 4 | 代理拒绝指定的用户名和密码 |
| mqtt.CONNACK\_REFUSED\_NOT\_AUTHORIZED | 5 | 用户名未经授权 |

**mqtt.client:lwt()**

设置最后的留言（选择）。一个代理将发布一个信息（qos=0、retain0，data=”offline”对注意/lwt），如果客户端不发送保活的包。

当最终将被发送给代理当连接的时候，lwt()必须被称为BEFORE在称为connect()。

代理将发布一个服务端最终的信息一旦注意到和服务端的连接被中断。代理将不会被注意到:当服务端没有发送保活包只要制定在mqtt.Client()-这个TCP连接被适当的关闭（在没有关闭mqtt之前）。这个代理尝试发送数据给服务端并且最终失败了，因为tcp连接不再被打开。

这意味着如果你指定了保活时间为120，仅仅是关闭了客户端设备并且代理也不发送任何数据给客户端，最终的消息也会被在关闭之后的120s发布。（发布完这个最后的消息，没有回应证明设别关闭——了解三次握手原则）。

**使用语法：**

mqtt:lwt(topic, message[, qos[, retain]])

**参数介绍：**

topic：将要发布消息的相应主题。

message：将要发布的消息（缓冲区的或者是字符串）。

qos：QOS等级，默认为0。

retain：保留标志位，默认为0。

**返回值：**

nil

**mqtt.client:on()**

给一个时间注册一个回调函数。

**使用语法：**

mqtt:on(event, function(client[, topic[, message]]))

**参数介绍：**

event：可以是：connect、message、offline、overflow

function(client[, topic[, message]]) ：回调函数。第一个参数是服务端。如果时间是message，那么，第二三个参数是被接收的主题和消息（字符串）。

**返回值：**

nil

**mqtt.client:publish()**

发布一条消息。

**使用语法：**

mqtt:publish(topic, payload, qos, retain[, function(client)])

**参数介绍：**

topic：将要发布消息的相应主题。

topic：将要发布消息的相应主题。

message：将要发布的消息（缓冲区的或者是字符串）。

qos：QOS等级。

retain：保留标志位。

function(client)：收到puback是激发的可选回调。注意：当唤醒publish()多次时，将为所有发布命令调用最后定义的回调函数。

**返回值：**

当成功时返回true，反之返回false。

**mqtt.client:subscribe()**

订阅第一个或者多个主题。

**使用语法：**

mqtt:subscribe(topic, qos[, function(client)]) mqtt:subscribe(table[, function(client)])

**参数介绍：**

topic：一个主题的字符串。

qos：QOS等级，默认为0。

table：一个订阅的’topic,qos’数组对。

function(client)：选择当返回订阅成功时的回调激励函数。注意：当订阅多次的时候，将为所有的订阅命令调用定义的最后一个回调函数。

**返回值：**

当成功时返回true，反之返回false。

**Example：**

*-- 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**)

小心：多次调用订阅的时候，你应该使用在上面的例子中的对个主题语法展现，如果你想要去同时订阅多个主题。

**mqtt.client:unsubscribe()**

对一个或者多个主题取消订阅。

**使用语法：**

mqtt:unsubscribe(topic[, function(client)]) mqtt:unsubscribe(table[, function(client)])

**参数介绍：**

topic：一个主题的字符串。

table：一个取消订阅的’topic,anything’数组对。

function(client)：选择当返回取消订阅成功时的回调激励函数。注意：当多次调用unsubscribe()函数时，为所有的取消订阅的命令调用被定义的最终回调函数。

**返回值：**

当成功时返回true，反之返回false。

**Example：**

*-- 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") **end**)

**net模块**

**TLS操作从TLS模块中已经移除。**

|  |  |
| --- | --- |
| Constants | 一个常量被用在其他函数中：net |
| net.createConnetction() | 创建一个客户端 |
| net.createServer() | 传建一个服务端 |
| net.CreateUDPSocket() | 传建一个UDP服务器 |
| net.multicastjoin() | 加入一个多播组 |
| net.multicastLeave() | 退出一个多播组 |
| net.server:close() | 关闭服务器 |
| net.server:listen() | 监听ID地址端口 |
| net.server:getaddr() | 返回服务器地址端口号 |
| net.socket:close() | 关闭服务器 |
| net.socket:connect() | 连接到一个远程的服务器 |
| net.socket:dns() | 提供主机名的一个DNS解析 |
| net.socket:getpeer() | 检索远程对等机的端口和IP。 |
| net.socket:getaddr() | 检索本地的服务器的地址和端口号 |
| net.socket:hold() | 通过放置组织TCP接收功能的请求来限制数据接收 |
| net.socket:on() | 给特别的事件注册回调函数 |
| net.socket:send() | 给远程对等机发送数据 |
| net.socket:ttl() | 改变或者检错在服务器的时间值 |
| net.socket:unhold() | 取消组织TCP接收数据，方法是撤销前面的hold() |
| net.udpsocket:close() | 关闭UDP服务器 |
| net.udpsocket:listen() | 监听IP地址的端口号 |
| net.udpsocket:on() | 给特殊事件注册回调函数 |
| net.udpsocket:send() | 给远程对等机发送数据 |
| net.upsocket:dns() | 给远程主机名提供DNS解析 |
| net.udpsocket:getaddr() | 检索本地服务器的IP地址和端口号 |
| net.udpsocket:ttl() | 改变或者检索服务器上的生存时间值 |
| net.dns.getdnsserver() | 获取DNS服务器的IP地址用于解析主机名 |
| net.dns.resolve() | 解析指定IP地址的主机名 |
| net.dns.setnsserver() | 设置DNS服务器的IP地址用于解析主机名 |

**Constants**

用在net.TCP和net.UDP函数中的常数。

**net.createConnection()**

创建一个客户端。

**使用语法：**

net.createConnection([type[, secure]])

**参数介绍：**

type：默认是net.TCP，也可以选择net.UDP

secure：1表示加密，0表示普通（默认）。

**注意：**

这将改变在即将到来的版本以至于net.createConnection将会一直创建未加密的TCP连接。

没有UDP连接，因为UDP是无连接的。由于没有连接类型参数应该被需要。然而对于UDP使用net.createUDPSocket()。为了创建一个安全的连接使用tls.createConnection()。

**返回值：**

对于net.TCP返回net.socket子模块。

对于net.UDPfanhui net.udpsocket子模块。

对于有安全模式的net.TCP返回tls.socket子模块。

**Example：**

net.createConnection(net.TCP, 0)

**net.createServer()**

创建一个服务端。

**使用语法：**

net.createServer([type[, timeout]])

**参数介绍：**

type：默认是net.TCP，也可以选择net.UDP

timeout：对于TCP服务器超时时间是1~28’8000秒，30秒是默认的（对于要断开连接的服务端）。

**注意：**

type参数将会被移除在将要到来的释放以至于net.createServer将总是创建一个基于TCP的服务器。然而对于DUP而言还是使用net.createUDSocket()。

**返回值：**

对于net.TCP返回net.socket子模块。

对于net.UDPfanhui net.udpsocket子模块。

**Example：**

net.createServer(net.TCP, 30) *-- 30s timeout*

**net.createUDPSocket()**

创建一个UDP服务器。

**使用语法：**

net.createUDPSocket()

**参数介绍：**

none（无）

**返回值：**

net.udpsocket sub module

**net.multicastJoin()**

加入一个多广播组。

**使用语法：**

net.multicastJoin(if\_ip, multicast\_ip)

**参数介绍：**

if\_ip：包含要加入多广播组的接口IP的字符串。”any”或者” ”影响所有的接口。

multicast\_ip：要加入的组。

**返回值：**

nil

**net.multicastLeave()**

退出一个多广播组。

**使用语法：**

net.multicastLeave(if\_ip, multicast\_ip)

**参数介绍：**

if\_ip：包含要加入多广播组的接口IP的字符串。”any”或者” ”影响所有的接口。

multicast\_ip：要退出的组。

**返回值：**

nil

**net.server模块**

**net.server:close()**

关闭服务器。

**使用语法：**

net.server.close()

**参数介绍：**

none（无）

**返回值：**

nil

**Example：**

*-- creates a server*

sv = net.createServer(net.TCP, 30)

*-- closes the server*

sv:close()

**net.server:listen()**

对指定的IP地址监听端口号。

**使用语法：**

net.server.listen([port],[ip],function(net.socket))

**参数介绍：**

port：端口号，可以忽略（如果忽略那么将会给一个随机的端口号）。

ip：IP地址字符串，可以忽略。

function(net.socket)：回调函数，如果连接成功作为参数传入一个函数。

**返回值：**

nil

**Example：**

*-- server listens on 80, if data received, print data to console and send "hello world" back to caller*

*-- 30s time out for a inactive client*

sv = net.createServer(net.TCP, 30)

**function** **receiver**(sck, data)

print(data)

sck:close()

**end**

**if** sv **then**

sv:listen(80, **function**(conn)

conn:on("receive", receiver)

conn:send("hello world")

**end**)

**end**

**net.server:getaddr()**

返回本地服务器的地址和端口。

**使用语法：**

net.server.getaddr()

**参数介绍：**

none

**返回值：**

port,ip（或者是nil,如果没有监听就是nil）

**net.socket模块**

**net.socket:close()**

关闭服务器。

**使用语法：**

close()

**参数介绍：**

none

**返回值：**

nil

**net.socket:close()**

连接一个远程服务器。

**使用语法：**

connect(port, ip|domain)

**参数介绍：**

port：端口号

ip：IP地址或者是域名

**返回值：**

nil

**net.socket:dns()**

给主机名提供DNS解析。

**使用语法：**

dns(domain, function(net.socket, ip))

**参数介绍：**

domin：域名

function(net.socket,ip)：回调函数。第一个参数是服务器，第二个参数是IP地址字符串。

**返回值：**

nil

**Example：**

sk = net.createConnection(net.TCP, 0)

sk:dns("www.nodemcu.com", **function**(conn, ip) print(ip) **end**)

sk = **nil**

**net.socket:getpeer()**

检索远程对等机的端口和IP地址。

**使用语法：**

getpeer()

**参数介绍：**

none

**返回值：**

port,ip（或者是nil,如果没有监听就是nil）

**net.socket:getaddr()**

检索本地服务器端口号和IP地址。

**使用语法：**

getaddr()

**参数介绍：**

none

**返回值：**

port,ip（或者是nil,如果没有监听就是nil）

**net.socket:hold()**

通过放置阻止TCP接收功能的请求来限制数据接收。这个请求不是立即有效的，Espressif建议当存储了到5\*1460字节的内存时候调用。

**使用语法：**

hold()

**参数介绍：**

none

**返回值：**

nil

**net.socket:on()**

给特殊的函数注册回调函数。

**使用语法：**

on(event, function())

**参数介绍：**

event：字符串，可以是”connection”,”reconnection”,”diconnection”,

”recevie”or”sent”。

function(net.socket[,string])：回调函数。可以是nil移除回调。

回调的第一个参数是服务器：

<1>如果事件是receive，第二个参数是收到数据作为参数。

<2>如果事件是：”reconnection”,”diconnection”,第二个参数是错误代码。

如果重新连接的时间是特定的，取消连接仅仅会收到”normal close”普通关闭事件。否则，所有连接错误（普通关闭）传入取消连接的事件。

**返回值：**

nil

**Example：**

srv = net.createConnection(net.TCP, 0)

srv:on("receive", **function**(sck, c) print(c) **end**)

*-- Wait for connection before sending.*

srv:on("connection", **function**(sck, c)

*-- 'Connection: close' rather than 'Connection: keep-alive' to have server*

*-- initiate a close of the connection after final response (frees memory*

*-- earlier here), https://tools.ietf.org/html/rfc7230#section-6.6*

sck:send("GET /get HTTP/1.1\r\nHost: httpbin.org\r\nConnection: close\r\nAccept: \*/\*\r\n\r\n")

**end**)

srv:connect(80,"httpbin.org")

注意：接收(receive)事件被取消对于任何的网络框架！因此，如果数据发送给设备超过1460字节（源自于以太网帧大小）他将取消多次。可能会有其他的场景接收数据被跨多个帧区分（例如，在multipart/form-data中的HTTP POST）。你需要手动缓冲数据，并找到方法来确定是否数据已经被就收到。

**local** buffer = **nil**

srv:on("receive", **function**(sck, c)

**if** buffer == **nil** **then**

buffer = c

**else**

buffer = buffer .. c

**end**

**end**)

*-- throttling could be implemented using socket:hold()*

*-- example: https://github.com/nodemcu/nodemcu-firmware/blob/master/lua\_examples/pcm/play\_network.lua#L83*

这是有例程代码的在官方的GitHub中。

**net.socket:send()**

发送远程对发送数据。

**使用语法：**

send(string[, function(sent)])

sck:send(data, fnA) 等价于sck:send(data) sck:on("sent", fnA)。

**参数介绍：**

string：在字符串中将要发送给服务器的数据。

function(sent)：发送字符串的额回调函数。

**返回值：**

nil

**注意：**

多个连续的send()回调不能被保证去工作（并且经常不会）作为网络的需要，被SDK作为分别任务对待。然而，订阅发送的事件在这个服务器并且发送额外的数据（或者被关闭的）在回调中。

**Example：**

srv = net.createServer(net.TCP)

**function** **receiver**(sck, data)

**local** response = {}

*-- if you're sending back HTML over HTTP you'll want something like this instead*

*-- local response = {"HTTP/1.0 200 OK\r\nServer: NodeMCU on ESP8266\r\nContent-Type: text/html\r\n\r\n"}*

response[#response + 1] = "lots of data"

response[#response + 1] = "even more data"

response[#response + 1] = "e.g. content read from a file"

*-- sends and removes the first element from the 'response' table*

**local** **function** **send**(localSocket)

**if** #response > 0 **then**

localSocket:send(table.remove(response, 1))

**else**

localSocket:close()

response = **nil**

**end**

**end**

*-- triggers the send() function again once the first chunk of data was sent*

sck:on("sent", send)

send(sck)

**end**

srv:listen(80, **function**(conn)

conn:on("receive", receiver)

**end**)

如果你不能保证你发送的数据都在内存中（记住响应是一个聚合），你可能使用这个明确的回调函数而不是建立一个表像这样的：

sck:send(header, **function**()

**local** data1 = "some large chunk of dynamically loaded data"

sck:send(data1, **function**()

**local** data2 = "even more dynamically loaded data"

sck:send(data2, **function**(sk)

sk:close()

**end**)

**end**)

**end**)

**net.socket:ttl()**

改变或者查阅服务器的时间值。

**使用语法：**

ttl([ttl])

**参数介绍：**

ttl：（选择）新的时间给保活值

**返回值：**

当前的值/新的ttl值

**Example：**

sk = net.createConnection(net.TCP, 0)

sk:connect(80, '192.168.1.1')

sk:ttl(1) *-- restrict frames to single subnet*

**net.socket:ttl()**

通过撤销先前的hold()组织TCP接收数据。

**使用语法：**

unhold()

**参数介绍：**

none（无）

**返回值：**

nil

**net.udpsocket 模块**

记住与TCP-UDP相比，它是无连接的。因此，存在一个较小的单自然地不匹配。然而你可以回调net.createConnnection()对于TCP，net.createUDPSocket()对于UDP。

其他的重要点：

<1>UDP服务器不能有连接回调对于监听函数。

<2>UDP服务器不能有连接函数。远程IP和端口号因此需要在send()函数中定义。

<3>UDP服务器的接收回调在data参数之后接收端口和IP。

**net. udpsocket:close()**

关闭UDP服务器。

这个语法和函数定义对于net.socket:close()

**net. udpsocket:listen()**

从指定的IP地址监听端口号。

这个语法和函数类似于net.server:listen()，但是不用提供回调参数。

**net. udpsocket:on()**

对指定的事件注册回调函数。

这个语法和函数类似于net.socket:on()。然而，仅仅”receive”,

”send”和”dns”被支持。

**注意：**

这个接收回调的端口号和IP在data参数之后。

**net. udpsocket:send()**

给指定的远程对发送数据。

**使用语法：**

send(port, ip, data)

**参数介绍：**

port：远程服务器端口。

ip：远程服务器IP。

data：要发送的有效负载。

**返回值：**

nil

**Example：**

udpSocket = net.createUDPSocket()

udpSocket:listen(5000)

udpSocket:on("receive", **function**(s, data, port, ip)

print(string.format("received '%s' from %s:%d", data, ip, port))

s:send(port, ip, "echo: " .. data)

**end**)

port, ip = udpSocket:getaddr()

print(string.format("local UDP socket address / port: %s:%d", ip, port))

在 \*nix系统能被测试被问题：

echo -**n** "foo" | nc -w1 -**u** <device-IP-address> 5000

**net. udpsocket:dns()**

对主机名提供DNS解析。

这个语法和函数定义对net.socket:dns()

**net. udpsocket:getddr()**

查询本地服务器的端口号和IP。

这个语法和函数跟net.socket:getddr()一样。

**net. udpsocket:ttl()**

改变或者查询服务器的时间值。

语法和函数与net.socket:ttl()一样。

**net.dns 模块**

**net. dns:getdnsserver()**

获取DNS服务器的IP地址用于解析主机名。

**使用语法：**

net.dns.getdnsserver(dns\_index)

**参数介绍：**

dns索引DNS服务器获取的范围是0-1。

**返回值：**

DNS服务器的IP地址（字符串）

**Example：**

print(net.dns.getdnsserver(0)) *-- 208.67.222.222*

print(net.dns.getdnsserver(1)) *-- nil*

net.dns.setdnsserver("8.8.8.8", 0)

net.dns.setdnsserver("192.168.1.252", 1)

print(net.dns.getdnsserver(0)) *-- 8.8.8.8*

print(net.dns.getdnsserver(1)) *-- 192.168.1.252*

**net. dns:resolve()**

对一个IP地址解析主机名。不需要一个服务器像net.socket.dns()。

**使用语法：**

net.dns.resolve(host, function(sk, ip))

**参数介绍：**

host：需要解析的主机名。

function(sk,ip)：当主机名被解析了。sk总是nil。

**返回值：**

nil

**Example：**

net.dns.resolve("www.google.com", **function**(sk, ip)

**if** (ip == **nil**) **then** print("DNS fail!") **else** print(ip) **end**

**end**)

**net. dns:setdnsserver()**

设置DNS服务器的IP地址用于解析主机名。默认是解析1.打开dns.com(208.67.222.222)。你能指定达到两个DNS服务器。

**使用语法：**

net.dns.setdnsserver(dns\_ip\_addr, dns\_index)

**参数介绍：**

dns\_ip\_addr：DNS服务器的IP地址。

dns\_index：DNS服务器设置(范围0-1)。因为它支持最大两个服务器。

**返回值：**

nil

**net.cert 模块**

这一部分可以参照TLS模块，链接保持向后兼容性。

**node 模块**

node模块提供了对系统电平的接口例如睡眠，重启或者是多种信息还有ID。

|  |  |
| --- | --- |
| node.bootreason() | 返回重置原因并且扩展重置详情 |
| node.chipid() | 返回ESP芯片的ID |
| node.compile() | 将Lua文本变异成Lua字节码，并保存为.Ic文件。 |
| node.dsleep() | 进入深度睡眠模式，当时间到达时唤醒 |
| node.dsleepMax() | 返回当前的最大的深度睡眠时长 |
| node.flashid() | 返回Flash芯片ID |
| node.flashindex() | 返回函数参考在Lua Flash储存中的一个文件 |
| node.flashreload() | 重载有Flash图片提供的Lua Flash储存 |
| node.flashsize() | 返回一个Flash芯片大小 |
| node.getcpufreq() | 获得当前CPU频率 |
| node.heap() | 返回当前可用的字节大小 |
| node.info() | 返回NodeMCU版本，芯片ID，FlashID，Flash大小，Flash大小，Flash模式，Flash速度和Lua文件储存使用静态。 |
| node.input() | 给Lua内部发送一个字符串 |
| node.output() | 将Lua解释器输出重定向到回到函数 |
| node.readvdd33()—deprecasted | 移动至ADC |
| node.restart() | 重启模块 |
| node.restore() |  |
| node.node.setcpufreq() | 改变CPU工作频率 |
| node.sleep() | 放置NodeMCU至轻睡眠模式 |
| node.stripdebug() | 控制节点期间保留的调试信息量 |
| node.osprint() | 控制是否调试输出从SDK被打印 |
| node.random() | 这个类似于数学中随机数 |
| node.egc.setmode() | 设置紧急垃圾收集模式 |
| node.egc.meminfo() | 返回在Lua运行的时候的内存使用情况 |
| node.task.post() | 使能Lua回调或者任务去post另一个任务请求 |

**node. bootreason()**

返回重置原因并且扩展重置信息。

第一个返回值是原始代码，不是在新SDK中介绍的新的重置信息代码，这个值是：

1. 电源开
2. 重置（软件）
3. 硬件重置通过重置键
4. WDT重置(看门狗超时)

第二个返回值是扩展的重置原因。值：

1. 电源开
2. 硬件看门狗重置
3. 异常重置
4. 软件看门狗重置
5. 软件重启
6. 唤醒深度睡眠
7. 扩展重置

通常，扩展重置是因为原始代码。原始代码仅向后兼容而保留。对于新的应用，建议使用这个扩展重启的原因。

扩展重置的一种情况是3，额外的值是返回包含了崩溃信息。有包括EXCCAUSE，EPC1，EPC2，EPC3，EXCVADDR，还有DEPC。

**使用语法：**

node.bootreason()

**参数介绍：**

none

**返回值：**

rawcode, reason [, exccause, epc1, epc2, epc3, excvaddr, depc ]

**Example：**

\_, reset\_reason = node.bootreason()

**if** reset\_reason == 0 **then** print("Power UP!") **end**

**node.chipid()**

返回ESP芯片ID。

**使用语法：**

node.chipid()

**参数介绍：**

none（无）

**返回值：**

芯片的ID（数字）。

**node.compile()**

将Lua文本变异成Lua字节码，并保存为.Ic文件。

**使用语法：**

node.compile("file.lua")

**参数介绍：**

filename：Lua文本文件名。

**返回值：**

nil

**Example：**

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()**

进入深度睡眠模式，到时唤醒。

最大的深度睡眠时长可以在node.dsleepMax()函数中查看，“对于ESP8266最大深度睡眠”声称最大是3.5小时。

**小心：**

这个函数只能在ESP8266PIN32(RST )和PIN8(XPD——DCDC aks GPIO16)被一同连接在一起使用sleep(0)将设置没有唤醒定时器，连接至GPIO至RST，芯片将会唤醒在第一个下降沿。

**使用语法：**

node.dsleep(us, option, instant)

**参数介绍：**

us：整数数字或者是nil，毫秒的睡眠时间，如果等于0，他将会永久睡眠，如果等于nil，将不会设置睡眠时间。

option：整数数字或者nil。如果nil，将会默认选择持续保持唤醒设置。

1. 初始化数据字节108。

>0，初始化字节108。

0，RF\_CAL或者不在深度睡眠之后唤醒，一来初始化字节108.

1. RF\_CAL深度睡眠后唤醒，会有很大的电流。
2. 不适用RF\_CAL在深度睡眠后唤醒，小电流。

4，不使能RF在深度失眠后唤醒，就像modem睡眠，最小电流。

instant：整数数字或者nil，如果存在且为不为0，芯片将立即进入深度睡眠并且将不等待切断WIFi。

**返回值：**

nil

**Example：**

*--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)

**node.dsleepMax()**

返回当前深度睡眠的时长。

**小心：**

当党指定了一个比理论的最大深度睡眠的时间长的时候，不建议超过最大值。在”Max deep seleep for ESP8266”这个文件中将不会唤醒，如果指定的睡眠时间超过了dsleepMax()。

注意：理论最大值有温度的影响:lower temp =shorter sleep 时长，反之时长越长。

**使用语法：**

max\_duration

**参数介绍：**

none（无）

**返回值：**

最大时长

**Example：**

node.dsleep(node.dsleepMax())

**node.flashid()**

返回Flash芯片的ID。

**使用语法：**

node.flashid()

**参数介绍：**

none（无）

**返回值：**

Flash的ID值（数字类型）

**node.flashindex()**

对于在Lua Flash中存储的函数返回一个函数参考。

**使用语法：**

node.flashindex(modulename)

**参数介绍：**

modulename：将要会被模块的名称。如果这个值是nil或者无效值，那么将会有一个信息列表被返回。

**返回值：**

如果LFS没有被加载，node.flashindex被估计为nil，后跟LFS的flash和映射基地址。

如果LFS被夹在并且函数被调用被在LFS文件中的模块，那么函数被用load()方法返回，并且另一个Lua也这样加载函数。

否则一个扩展的信息列表被返回：LFS的Unix时间，这个flash和应涉及地质还有当前的额长度，还有一个在Lua文件中的模块的名字的数组。

**Example：**

这个node.flashindex()是低版本的API，通常使用标准得Lua代码包装，呈现一个更简单的应用程序API。请参考\_init.lua模块的lua\_examples/lfs文件夹怎么做。

**node.flashreload()**

重新加载有Flash镜像的LFS文件。Flash镜像使用了luac.cross命令主机机器产生。

**使用语法：**

node.flashreload(imageName)

**参数介绍：**

imageName：将要加载在LFS中的镜像文件的名字。

**返回值：**

Error message：LFS镜像现在的被压缩。imagename是一个有效的LFS镜像，这个扩展并且加载进flash中。这个ESP之后会立即重新启动。所以在成功加载的情况下控件不返回调用的程序。重加载将会传入两个文件；并且在第一个数据文件和一个格式头文件还有所有的错误。如果检测到有错误将会返回。

**node.flashsize()**

返回flash芯片的字节大小。在4MB模块，像ESP12返回的值是4194304=4096KB。

**使用语法：**

node.flashsize()

**参数介绍：**

none（无）

**返回值：**

flash的字节大小（数字）

**node.getcpufreq()**

获取当前的CPU频率。

**使用语法：**

node.getcpufreq()

**参数介绍：**

none（无）

**返回值：**

当前的CPU频率（数字类型）

**Example：**

**do**

**local** cpuFreq = node.getcpufreq()

print("The current CPU frequency is " .. cpuFreq .. " MHz")

**end**

**node.heap()**

返回当前可用的字节大小。注意由于碎片化的原因，真实分配的大小可能没有那么大。

**使用语法：**

node.heap()

**参数介绍：**

none（无）

**返回值：**

系统剩余空间大小（数字）。

**node.info()**

返回NodeMCU的版本信息，芯片ID，FlashID，flash大小，flash模式，flash速度，并且Lua文件存储使用。

**使用语法：**

node.info()

**参数介绍：**

none（无）

**返回值：**

majorVer：（数字类型）

minorVer：（数字类型）

devVer：（数字类型）

chipid：（数字类型）

flashid：（数字类型）

flashsize：（数字类型）

flashmode：（数字类型）

flashspeed：（数字类型）

**Example：**

majorVer, minorVer, devVer, chipid, flashid, flashsize, flashmode, flashspeed = node.info()

print("NodeMCU "..majorVer.."."..minorVer.."."..devVer)

**node.input()**

发送一个字符串到Lua内部。类似于pcall(loadstring(str))，但是没有单行的限制。

**注意：**

这个函数仅仅在回调函数中有效。如果直接在控制台使用他将不能使用。

**使用语法：**

node.input(str)

**参数介绍：**

str：Lua块。

**返回值：**

nil

**Example：**

sk:on("receive", **function**(conn, payload) node.input(payload) **end**)

**node.output()**

将Lua解释器输出重定向到回调函数。也可以选择在控制台输出。

**小心：**

不要尝试print()或者或者Lua内部的输出在回调函数中。这样做会导致无线递归，并导致看门狗触发重新启动。

**使用语法：**

node.output(function(str), serial\_debug)

**参数介绍：**

output\_fn(str)：一函数接收每个打印的字符串，并且可以发送至输出服务器（或者是文件）。

serial\_debug：1个输出也会战术。0没有输出。

**返回值：**

nil

**Example：**

**function** **tonet**(str)

sk:send(str)

**end**

node.output(tonet, 1) *-- serial also get the Lua output.*

*-- a simple telnet server*

s=net.createServer(net.TCP)

s:listen(2323,**function**(c)

con\_std = c

**function** **s\_output**(str)

**if**(con\_std~=**nil**)

**then** con\_std:send(str)

**end**

**end**

node.output(s\_output, 0) *-- re-direct output to function s\_ouput.*

c:on("receive",**function**(c,l)

node.input(l) *-- works like pcall(loadstring(l)) but support multiple separate line*

**end**)

c:on("disconnection",**function**(c)

con\_std = **nil**

node.output(**nil**) *-- un-regist the redirect output function, output goes to serial*

**end**)

**end**)

**node.readvdd33()—deprecated**

移动至adc.readvdd33()。

**node.restart()**

重启芯片。

**使用语法：**

node.restart()

**参数介绍：**

none（无）

**返回值：**

nil

**node.restore()**

使用SDK函数sysytem\_restore()将系统配置还原成默认，文档中描述如下：

API重置默认值：wifi\_station\_set\_auto\_connext,

wifi\_set\_phy\_mode,wifi\_softap\_set,wifi\_station\_set\_config,wifi\_set\_opmode,还有API的信息记录#define Ap\_CACHE。

**使用语法：**

node.restore()

**参数介绍：**

none（无）

**返回值：**

nil

**Example：**

node.restore()

node.restart() *-- ensure the restored settings take effect*

**node.setcpufreq()**

改变CPU的工作频率。

**使用语法：**

node.setcpufreq(speed)

**参数介绍：**

speed：常量node.CPU80MHZ或者是node.CPU160MHZ。

**返回值：**

CPU指定频率（数字类型）。

**Example：**

node.setcpufreq(node.CPU80MHZ)

**node.sleep()**

将NodeMCU设置为轻睡眠模式减少电流的消耗。

如果WiFi被中断，NodeMCU不能进入轻睡眠模式。

所有的激活时间都被中断并且之后重现当唤醒NodeMCU。

**注意：**

这不是默认的。请在app/include/user\_config.h中PMSLEP\_ENABLE使能它。

**使用语法：**

node.sleep({wake\_pin[, int\_type, resume\_cb, preserve\_mode]})

**参数介绍：**

wake\_pin：1-12引脚去连接唤醒中断。注意Pin0（GPIO16）不支持中断。

请参考GPIO模块了解更多的信息。

int\_type：中断的类型你想要唤醒Node的。（默认是node.INT\_LOW）

有效中断模式：

node.INT\_UP：上升沿。

node.INT\_DOWN：下降沿。

node.INT\_BOTH：双边沿。

node.INT\_LOW：低电平。

node.INT\_HIGH：高电平。

resume\_cb：当从中断中唤醒时执行的回调（选择）。

preserve\_mode：通过接点水面保持当前WiFi模式（可选，默认真）：

如果是真，那么Station和StationAP模式将会自动重新连接到当前的配置接入点当NodeMCU。

如果是false，注销WiFi模式并且身下NodeMCU在wifi.NULL\_MODE。WiFi模式将被储存去初始模式重新开始。

**返回值：**

nil

**Example：**

*--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)

**node.stripdebug()**

在node.compile()过程中控制debug信息，并且允许从已经存在的编译成功的Lua代码中移除debug信息。

只建议高级用户使用，NodeMCU默认适用于所有用户。

**使用语法：**

node.stripdebug([level[, function]])

**参数介绍：**

level：

1. 不取消debug信息。
2. 取消本地和上值得debug信息。
3. 取消本地和上值还有行号debug信息。

function：不允许按照selfnv除去0以外的编译函数。

如果没有参数被赋值，那么当前的默认值设置将被返回。如果函数是省略，这个默认的设置对将来的编译。函数的参数谁用规则类似于setfenv()。

**返回值：**

如果没有参数调用，那么返回当前的设置。否则nil被返回。

**Example：**

node.stripdebug(3)

node.compile('bigstuff.lua')

**node.osprint()**

控制从安信可SDK的解调输出被打印。注：如果固件是用开发的工具制成的，这是唯一可行的。

**使用语法：**

node.osprint(enabled)

**参数介绍：**

enable：这个也是true去允许输出，或者false不能输出。默认是false。

**返回值：**

Nothing

**Example：**

node.osprint(**true**)

**node.random()**

这个函数类似于math.random除了他是用真的随机数产生从ESP8266硬件中。他返回一个所需要范围内的均匀分布的数字。他也可以获得超过范围的值。

它可以被调用用三种方式。在Floating中NodeMCU没有参数，它返回一个随机数在0-1区间中随机分布的。当只有一个参数的时候，一个整数n，他将返回一个整数随机数x例如1<=x<=n。例如你可以模拟这个结果是random(6)，最终，random能够被调用两个整数参数，我和你，去获得一个随机的整数x，例如i<=x<=u。

**使用语法：**

node.random() node.random(n) node.random(l, u)

**参数介绍：**

n：整数的数字参数能够被返回的额，在1范围内的n

l：最小的边界

u：最大的边界

**返回值：**

一个随机数在适当区间的。注意0参数将总是返回0在integer建立的时候。

**Example：**

print ("I rolled a", node.random(6))

**node.egc 模块**

**node.egc.setmode()**

设置紧急垃圾回收模式。这个EGC白名单提供了更多的细节对于EGC。

**使用语法：**

node.egc.setmode(mode, [param])

**参数介绍：**

mode：

node.egc.NOT\_ACTIVE：EGC不活动，在内存不足的情况下不会强制执行任何收集循环。

node.egc.ON\_ALLOC\_FAILURE：尝试去分配新的储存块，并且执行回收如果分配失败的话。如果这个分配失败甚至执行回收，这个分配将会返回失败。

node.egc.ON\_MEM\_LIMIT：执行回收当内存被Lua脚本执行超过了顶层限制。如果顶层限制不能被处理，甚至之后执行回收，这个分配将会返回错误。如果返回限制是消极的，他在解释为保持对量。无论什么时候自由的对（作为node.heap()有请求的限制，这个回收被执行）。

node.egc.ALWAYS：执行这个回收再内存分配之前。如果这个分配失败甚至之后执行回收，这个分配返回错误。这个模式飞扬搞笑对于内存的保护，但是是最慢的。

level：在node.egc.ON\_MEM\_LIMIT中，指定了内存的限制。

**返回值：**

nil

**node.egc.meminfo()**

返回在Lua执行时的内存使用情况。

**使用语法：**

total\_allocated, estimated\_used = node.egc.meminfo()

**参数介绍：**

none

**返回值：**

total\_allocated：在Lua执行时总的分配字节数。这是使用node.egc.on\_mem\_limit选项时的相关数字，其值为正值。

estimated\_used：这个值显示了建立时分配的内存。

**node.task 模块**

**node.task.post()**

启用lua回调或任务以发布另一个任务请求。注意每一个例子有多个任务可以被发布在任何的任务中，但是高等级的传输最快。

如果任务队列是满的那么一个满队列的错误会暴露。

**使用语法：**

node.task.post([task\_priority], function)

**参数介绍：**

task\_priority：（选项）

node.task.LOW\_PRIORITY=0

node.task.MEDIUM\_PRIORITY=1

node.task.HIGH\_PRIORITY=2

function：回调函数执行当任务执行的时候。

如果忽略的话，那么默认设置为node.task.MEDIUM\_PRIORITY。

**返回值：**

nil

**Example：**

**for** i = node.task.LOW\_PRIORITY, node.task.HIGH\_PRIORITY **do**

node.task.post(i,**function**(p2)

print("priority is "..p2)

**end**)

**end**

输出：

priority **is** 2

priority **is** 1

priority **is** 0

**PWM 模块**

|  |  |
| --- | --- |
| pwm.close() | 让指定的GPIO引脚退出PWM模式 |
| pwm.getclock() | 获取选定的PWM引脚频率 |
| pwm.getduty() | 获取选定的引脚占空比 |
| pwm.setclock() | 设置PWM频率 |
| pwm.setduty() | 设置引脚的占空比 |
| pwm.setup() | 将引脚设置成PWM模式 |
| pwm.start() | PWM开始，将波形应用于引脚 |
| pwm.stop() | 暂停PWM输出波形 |

**pwm.close()**

对指定的引脚退出PWM模式。

**使用语法：**

pwm.close(pin)

**参数介绍：**

pin：1-12引脚，查阅引脚索引图。

**返回值：**

nil

**pwm.getclock()**

获得指定的引脚的PWM频率。

**使用语法：**

pwm.getclock(pin)

**参数介绍：**

pin：1-12引脚，查阅引脚索引图。

**返回值：**

数字：引脚的PWM频率。

**pwm.getduty()**

获取选定引脚的占空比。

**使用语法：**

pwm.getduty(pin)

**参数介绍：**

pin：1-12引脚，查阅引脚索引图。

**返回值：**

数字：占空比，最大1023。

**pwm.setclock()**

设置PWM的频率。注意：脉宽调制频率的设置也将同步改变其他设置，如果有的话。只有一个PWM频率能被允许在这个系统中。

**使用语法：**

pwm.setclock(pin, clock)

**参数介绍：**

pin：1-12引脚，查阅引脚索引图。

clock：1~1000，PWM的频率。

**返回值：**

nil

**pwm.setduty()**

设置引脚占空比。

**使用语法：**

pwm.setduty(pin, duty)

**参数介绍：**

pin：1-12引脚，查阅引脚索引图。

duty：1~1023，PWM的占空比，最大1023（10字节）。

**返回值：**

nil

**Example：**

*-- 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()**

设置引脚的至PWM模式。最多只能将6个引脚设置为脉宽调制模式。

**使用语法：**

pwm.setup(pin, clock, duty)

**参数介绍：**

pin：1-12引脚，查阅引脚索引图。

clock：1~1000，PWM的频率。

duty：1~1023，PWM的占空比，最大1023（10字节）。

**返回值：**

nil

**Example：**

*-- set pin index 1 as pwm output, frequency is 100Hz, duty cycle is half.*

pwm.setup(1, 100, 512)

**pwm.start()**

PWM开始，将波形应用至GPIO引脚。

**使用语法：**

pwm.start(pin)

**参数介绍：**

pin：1-12引脚，查阅引脚索引图。

**返回值：**

nil

**pwm.stop()**

停止PWM波形输出。

**使用语法：**

pwm.stop(pin)

**参数介绍：**

pin：1-12引脚，查阅引脚索引图。

**返回值：**

nil

**SJSON 模块**

此模块支持JSON模块。允许编码至JSON格式也可以从JSON格式解码。

请注意嵌套的表格需要大量的内存进行编码。以至于报错，使用pcall()。

这个代码使用了JSON的字节流库能够分析JSON。

此模块可以在两个方面使用。简单的方面是使用他直接转换成JSON格式（你可以使用\_G .cjson=sjson）。更高级的方法是使用流接口。这允许编码并且译码更大的对象。

JSON空值的处理如下：

默认的，解码器将null代表为sjson.NULL（这是一个userdata对象）。这个是CJSON的行为。

编码器总是转换任何的用户数据对象至null。

选择性的，一个单字符串可以被指定在编码器和解码器。这个字符串在编码/解码到指定的JSON-null值。这个字符串应该被用在任何地点个在你的数据接口中。’\0’被允许。

当斑马Lua对象的时候，如果发现函数，那么它被调用（没有参数）并且单返回值将会被编码至这个函数中。

**注意：**

下面的所有示例都使用内存中的JSON或从SPiffs文件系统读取的内容。然而，流接口真正闪光的地方是从远程资源中获取大型的JSON结构并实时提取值。详细的流式示例可以在/lua\_examples文件夹中找到。

|  |  |
| --- | --- |
| sjson.encoder() | 这个创建了一个编码对象能够转换一个Lua对象至JSON编码字符串。 |
| sjson.encoder:read | 这个获取了JSON编码后的数据 |
| sjson.encode() | 编码Lua表至JSON字符串 |
| sjson.decoder() | 这时的译码对象能解析JSON编码字符串值Lua对象 |
| sjson.decoder:write | 提供更多被解析的数据至Lua对象 |
| sjson.decoder:result | 获取了译码Lua对象，或者如果译码工作不能完成将会引起错误 |
| sjson.decode() | 解码一个Lua对象至Lua表 |
| Constants | 一个常量，sjosn |

**sjson.encoder()**

这创建了一个编码对象，能够转换一个Lua对象到JSON编码字符串。

**使用语法：**

sjson.encoder(table [, opts])

**参数介绍：**

table：需要编码的数据。

opts：可选选项表：

depth：最大的编码深度需要编码的表。默认是20，一般可以覆盖基本的普通操作。

null：把字符串比成null。

**返回值：**

sjson.encoder对象。

**sjson.encoder:read**

获取一块JSON编码数据。

**使用语法：**

encoder:read([size])

**参数介绍：**

size：选择需要返回的字节大小。默认是1024。

**返回值：**

最大为字节大小的字符串，如果编码完成并且返回了所有数据，则为零。

**Example：**

下面的例子输出了64字节的块，一个JSON的编码字符串包含了每一个文件的第一个4k在文件系统中。总共的字符串的数量可能比NodeMCU的内存还大。

**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()**

编码一个Lua表格至JSON字符串格式。这时间便的方法停提供了与Sjson向后兼容。

**使用语法：**

sjson.encode(table [, opts])

**参数介绍：**

table：需要编码的数据。

opts：可选选项表：

depth：最大的编码深度需要编码的表。默认是20，一般可以覆盖基本的普通操作。

null：把字符串比成null。

**返回值：**

JSON字符串。

**Example：**

ok, json = pcall(sjson.encode, {key="value"})

**if** ok **then**

print(json)

**else**

print("failed to encode!")

**end**

**sjson.decoder()**

这个制作了一个译码对象能够解析JSON编码的字符串至Lua对象。一个元表能被指定多有的新创建的Lua表。这允许你可以处理任何的值，当他插入值任何的表中（通过接入\_\_newindex方法）。

**使用语法：**

sjson.decoder([opts])

**参数介绍：**

opts：可选选项表：

depth：最大的编码深度需要编码的表。默认是20，一般可以覆盖基本的普通操作。

null：把字符串比成null。

metatable：要用作返回对象中所有新表的元表的表。

**返回值：**

sjson.decoder对象。

**Metatable：**

当调用这个metetable【元表】(如果它存在的话)这里有两个原则。

\_\_newindex：无论什么时候新表被创建的时候，这个都是一个标准得方法调用。

checkpath：无论什么时候新表被创建，那么它被调用（如果被定义）。当他被调用的时候有两个参数：

table：一个新表被创建。

path：这是一个根目录中的列表键。如果这个对象是在结果中返回，那么返回true，反之返回false。

例如，当译码{“foo”:[1,2,[]]}这个checkpath参数将会被调用像下面的：

checkpath({},{})，table参数是将与JSON对象的值对应的对象。

checkpath({},{“foo”})，table参数是将与外部JSON数组的值对应的对象。

checkpath({},{“foo”,3})table参数是将于空的内部JSON数组对应的对象。

当这个checkpath方法被调用时候，这个元表已经接入了新表。因此这个checkpath方法能够代替它。例如，如果你正在解码

{“foo”:{“bar”:[1,2,3,4],”cat”:[5]}}并且，由于一些原因，你不想补货bar的值，那么有多种方法可以实现：

在这个新的\_\_newindex元表中，仅仅查看键的值，如果键的值是”bar”那么跳过rawset。只有你想跳过所有的”bar ”键时，这才有效。在checkpath方法中，如果路径是[“foo”]，则返回false。

使用下面的checkpath：checkpath=function(tab,path)

tab[‘\_\_json\_path’]=path return ture end。这将会保存这个路径在每一个对象中。现在\_\_newindex方法，能够展现更多的精密过滤。

之所以能够过滤，是因为他能够在内存受限的平台上处理非常大的JSON响应。许多的API返回大量的信息，这将超出平台的内存预算。例如：

https://api.github.com/repos/nodemcu/nodemcu-firmware/contents超过13K，但是如果你只需要download\_url的键，那么总共大小大约600B。这能处理用一个简单的\_\_newindex方法。

**sjson.decoder:write**

这提供更多的数据可以被解码至Lua对象。

**使用语法：**

decoder:write(string)

**参数介绍：**

string：下一段JSON编码的数据

**返回值：**

构造的Lua对象或者nil(如果解码没有完成)

**ERROR：**

如果在解码过程中发生解码错误，那么一个错误将会抛出并且终止分析。这个不能被再次使用。

**sjson.decoder:result**

获取了解码Lua对象，或者引起一个错误如果这个解码没有完成。这能被调用多次并且返回同一时间返回相同的对象。

**使用语法：**

decoder:result()

**ERROR：**

如果解码没有完成，将会抛出错误。

**Example：**

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**

下一个例子方面使用了元表参数。在这个情况下他只能输出操作，如果可以的话，可以终止任务。

**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()**

解码一个JSON字符串至Lua表。这个方便的一种方法对于cjson向后兼容。

**使用语法：**

sjson.decode(str[, opts])

**参数介绍：**

str：需要解码的JSON字符串。

opts：可选选项表：

depth：最大的编码深度需要编码的表。默认是20，一般可以覆盖基本的普通操作。

null：把字符串比成null。

metatable：要用作返回对象中所有新表的元表的表。

**返回值：**

用JSON数据表示的Lua表。

**ERROR：**

如果字符串不是有效的JSON格式的话，那么将会抛出异常。

**Example：**

t = sjson.decode('{"key":"value"}')

**for** k,v **in** pairs(t) **do** print(k,v) **end**

**Constants**

这是一个常量，sjson.NULL格式的，它一般被用于Lua结构表示JSON是null的时候。

**SNTP 模块**

这根SNTP模块接入了简单网络获取服务端。这个模块支持任何选播的NTP模块，如果支持NTP服务器在你的网络中，那么你不必要知道NTP服务器的IP地址。默认的他将用这个0.nodemcu.pool.ntp.org或者是3.nodemcu.pool.ntp.org。这些服务器基本上适用于所有的包。

当和rtctime模块一起编译的时候，它还提供与IT的无缝集成，潜在地减少了获取NTP同步时钟的进程到sntp.sync()没有参数的情况下被调用。

|  |  |
| --- | --- |
| sntp.sync() | 尝试去获取同步时钟 |
| sntp.setoffset() | 设置RTC时钟和NTP时钟的偏移量 |
| sntp.getoffset() | 获取RTC时钟个NTP时钟的偏移量 |

**sntp.sync()**

尝试获取同步时钟。

对于醉最佳的结果，你可能想要定期调用次函数以补偿内部时钟漂移。对于这个稳定的RTC时钟模块的文件它允许同步定时器深度睡眠并且它是必要的在WiFI初始化之后添加至init.lua文件中模块重置。注意，一个单服务器可以提供一个参数（名字或者地址），或者是一个可以被提供的服务器的列表。

如果所有的支持域名/地址是有效的，那么错误的回调将会调用在参数类型1。否则，如果有至少一个有效的名字或者地址，那么sync将会被展示。

如果任何的sync操作失败，（可能这个设备没有连接到网络中），那么所有的将会重新查找一次。

**使用语法：**

sntp.sync([server\_ip], [callback], [errcallback], [autorepeat]) sntp.sync({ server1, server2, .. }, [callback], [errcallback], [autorepeat])

**参数介绍：**

server\_ip：如果非空，服务器将会被使用。如果是空，那么之前的连接的服务器被使用。如果之前没有服务器，那么这个ntp池中的服务器被使用。如果任何的选播的服务器被使用，那么这第一个回应的服务器将会被使用。

server1,server2：一个或者多个尝试的ip地址，或者是dns名字。

callback：如果提供，那么将会调用一个成功的同步是孩子能够，有四个参数，secends,microseconds,server和info。注意当rtctime模块被接入，不需要明确的调用rtctime.set()-这个模块小心使用内部自动化，对于最好的精确。这个info的参数是一个semi的表，下面将会对他进行描述。

ercallback：两个参数失败调用。第一个参数是内部描述的错误类型。在放弃并且报告错误之前，模块会自动执行多次错误。这个秒是一个字符串包含了支持的信息，错误代码：

1：DNS查找失败（第二个参数是失败的DNS名称）。

2：内存分配失败。

3：UDP发送失败。

4：时间超时，没有接收到NTP回应。

autorepeat：如果这是一个非空值，那么这个同步时钟将会发生每1000秒并且尝试如果可以的话调整时钟。这个回调江湖调用在每个sync操作之后。

**返回值：**

nil

**Info table（详情列表）：**

这个传入了成功调用并且包含有用的信息关于时间同步的在完成的时候。表中的关键词有：

offset\_s：这是一个选项并且包含了秒的数字时钟被调整的。这也仅仅展现了非常大（大量的数字）的调整。典型的，这是仅仅展现了初始时钟回调。

offset\_us：这是一个选项（但是offset\_s和sffset\_us总是存在）。这包含了毫秒的信息时钟被调整的。

delay\_us：这是到服务器的延迟返回秒。此设置不确定度略小于此值。

stratum：服务器层次。

leap：包含了来自NTP协议的跳跃位。0表示没有挂起的闰秒，1表示在UTC月底挂起的额外闰秒，2表示在UTC月底挂起的闰秒删除。

**Example：**

*-- Use the nodemcu specific pool servers and keep the time synced forever (this has the autorepeat flag set).*

sntp.sync(**nil**, **nil**, **nil**, 1)

*-- Single shot sync time with a server on the local network.*

sntp.sync("224.0.1.1",

**function**(sec, usec, server, info)

print('sync', sec, usec, server)

**end**,

**function**()

print('failed!')

**end**

)

**sntp.setoffset()**

setoffset调用启用显式的闰秒跟踪，并使rtc时钟更均匀地滴答——但它与墙壁时钟时间不同步。秒数是偏移量。

**使用语法：**

sntp.setoffset([offset])

**参数介绍：**

offset：在RTC时钟和NTP时钟之间的偏移量。这可以忽略，并且默认是0。这个偏移量可以被追踪。

**返回值：**

nil

**sntp.getoffset()**

获取RTC时钟和NTP时钟之间的偏移量。这个值应该是从RTC时钟获取NTP时钟——与墙时钟通信。如果偏移量被之前调用的数值改变了，那么在这其中有了闰秒。

**使用语法：**

offset = sntp.getoffset()

**返回值：**

返回当前的偏移量。

**Timer 模块**

定时器模块允许接入简单的定时器和系统计时器还有正常运行时间。

目的是设置常规的任务，超时操作，还有提供低分辨率的三角。

然而，定时器模块不是计时模块。当大多数会超过毫秒甚至是微秒，这个精确度时候到限制，并且有多种错误导致不准确的时间保持。考虑将RTCTime模块用于“墙时钟”时间。

**注意：**

NodeMCU之前提供了7个静态的定时器，标号为0-6，这些可以被用于之前的API定时器初始化，使用tmr.create()方法。之后很长时间的考虑，这些在2019年第一季度的时候被移除。

|  |  |
| --- | --- |
| tmr.create() | 创建一个动态时钟对象；请看下面它的方法使用表 |
| tmr.delay() | 对于指定的毫秒数字进行忙循环 |
| tmr.now() | 返回系统计时器，毫秒计时 |
| tmr.softwd() | 提供一个软件看门狗，需要去重新武装或者在到期前禁用，否则系统将重新启动 |
| tmr.time() | 返回系统正常运转的时间，秒级别 |
| tmr.wdclr() | 喂看门狗 |
| Timer Object Methods |  |

**tmr.create()**

创建一个动态定时器对象；请参照下面的方法表。

在控制函数中可以使用动态动态计时器代替数字ID。也可以在面向对象方式中控制它。

在定时器对象中被支持的函数：

t:alarm()

t:interval()

t:register()

t:state()

t:stop()

t:unregister()

**参数介绍：**

none（无）

**返回值：**

timer：对象。

**Example：**

**local** mytimer = tmr.create()

mytimer:register(5000, tmr.ALARM\_SINGLE, **function** (t) print("expired"); t:unregister() **end**)

mytimer:start()

**tmr.delay()**

对于指定的毫秒数字进行忙循环

这是一种不好的方式，因为在这期间没有任何任务在运行，并且网络栈（和其他的东西）都可能崩溃。唯一适合使用的时间tmr.delay()可能是处理命令之间需要（非常）短暂延迟的外围设备或类似设备。请小心使用！

由于时间不准确以及在此期间可能运行的中断，延迟的实际时间量可能会显著增加。

**使用语法：**

tmr.delay(us)

**参数介绍：**

us：忙循环的毫秒数。

**返回值：**

nil

**Example：**

tmr.delay(100)

**tmr.now()**

返回系统计数器，毫秒级。限制31字节，超过之后会重新至0.如果要使用此函数接触缓冲或者限制GPIO输入，这是非常重要的。

**使用语法：**

tmr.now()

**参数介绍：**

none

**返回值：**

当前系统计数器的值。

**Example：**

print(tmr.now())

print(tmr.now())

**tmr.softwd()**

提供简单的软件看门狗，需要重新武装或者在他到期前禁用，否则系统会重新启动。

**使用语法：**

tmr.softwd(timeout\_s)

**参数介绍：**

timeout\_s：看门狗超时时间，秒位。如果想要禁用看门狗的话，将值附为-1或者是其他的负数。

**返回值：**

nil

**Example：**

**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()**

返回系统的正常运行的时间。限制为31字节，在到达上限时自动清零。

**使用语法：**

tmr.time()

**参数介绍：**

none

**返回值：**

系统正常运行的时间，秒级，可能会被包围。

**Example：**

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

**tmr.wdclr()**

喂系统看门狗。

通常情况下，如果你需要使用这个函数，会出错。

这个NodeMCU的事件处理模块意味着不需要去设置硬件的循环等到事务的发生。相反，只需使用回调在发生某些事情时得到通知。有了这个方法，应该永远都不用喂看门狗了。

**使用语法：**

tmr.wdclr()

**参数介绍：**

none

**返回值：**

nil

**Timer Object Methods（对象方法）**

**tobj:alarm()**

这是一个便利的函数包括了tobj:register()和tobj:start()在这一个调用语句中。

为了能释放定时器资源当使用完成后，请调用tobj:unregister()。对于一次性计时器来说是没有必要的，除非他们在到期前被停止。

**使用语法：**

tobj:alarm(interval\_ms, mode, func())

**参数介绍：**

interval\_ms：定时器间隔毫秒级。最大数值：6780947（1:54:30.947）。

mode：定时器模式。

tmr.ALARM\_SINGLE：一次性时钟（不需要调用unresgister）

tmr.ALARM\_SEMI：手动重复时钟（调用start()去重新开始）。

tmr.ALARM\_AUTO：自动重复性时钟。

func(timer)：回调函数，被作为一个定时器对象调用。

**返回值：**

如果定时器重新开始则返回true，反之返回false。

**Example：**

**if** **not** tmr.create():alarm(5000, tmr.ALARM\_SINGLE, **function**()

print("hey there")

**end**)

**then**

print("whoopsie")

**end**

**tobj:interval()**

改变注册定时器的有效间隔。

**使用语法：**

tobj:interval(interval\_ms)

**参数介绍：**

interval\_ms：新的定时器间隔毫秒级的。最大数值：6870947(1:54:30.947)。

**返回值：**

nil

**Example：**

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()**

配置定时器并且注册回到函数在定时器到期时调用。

为了在使用完之后释放定时器资源，请调用tobj:unresgister()。一次性定时器没有必要调用，除非在到期限停止使用定时器。

**使用语法：**

tobj:register(interval\_ms, mode, func())

**参数介绍：**

interval\_ms：定时器间隔毫秒级。最大数值：6780947（1:54:30.947）。

mode：定时器模式。

tmr.ALARM\_SINGLE：一次性时钟（不需要调用unresgister）

tmr.ALARM\_SEMI：手动重复时钟（调用start()去重新开始）。

tmr.ALARM\_AUTO：自动重复性时钟。

func(timer)：回调函数，被作为一个定时器对象调用。

注意：请在没有开启前注册定时器。

**返回值：**

nil

**Example：**

mytimer = tmr.create()

mytimer:register(5000, tmr.ALARM\_SINGLE, **function**() print("hey there") **end**)

mytimer:start()

**tobj:start()**

开始或者重新开始之前配置的定时器。

**使用语法：**

tobj:start()

**参数介绍：**

none（无）

**返回值：**

如果定时器开始执行则返回true，反之返回false。

**Example：**

mytimer = tmr.create()

mytimer:register(5000, tmr.ALARM\_SINGLE, **function**() print("hey there") **end**)

**if** **not** mytimer:start() **then** print("uh oh") **end**

**tobj:state()**

检查定时器的状态。

**使用语法：**

tobj:state()

**参数介绍：**

none（无）

**返回值：**

（bool，int）或者nil

如果指定的定时器被注册，返回是否现在是否开始还有当前的模式。如果定时器没有被注册，则返回nil。

**Example：**

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()**

停止正在运行的定时器，但是不会对它取消注册。一个被停止的定时器可以使用tobj:start()重新开启。

**使用语法：**

tobj:stop()

**参数介绍：**

none（无）

**返回值：**

如果定时器停止了则返回true，反之返回false。

**Example：**

mytimer = tmr.create()

**if** **not** mytimer:stop() **then** print("timer not stopped, not registered?") **end**

**tobj:unregister()**

停止当前正在运行的定时，并且取消相关联回调。

这个函数对于一次性的定时器是不必要的（tmr.ALARM\_SINGLE），因为当使用完成（解雇）之后对他们进行取消注册。

**使用语法：**

tobj:unregister()

**参数介绍：**

none（无）

**返回值：**

nil