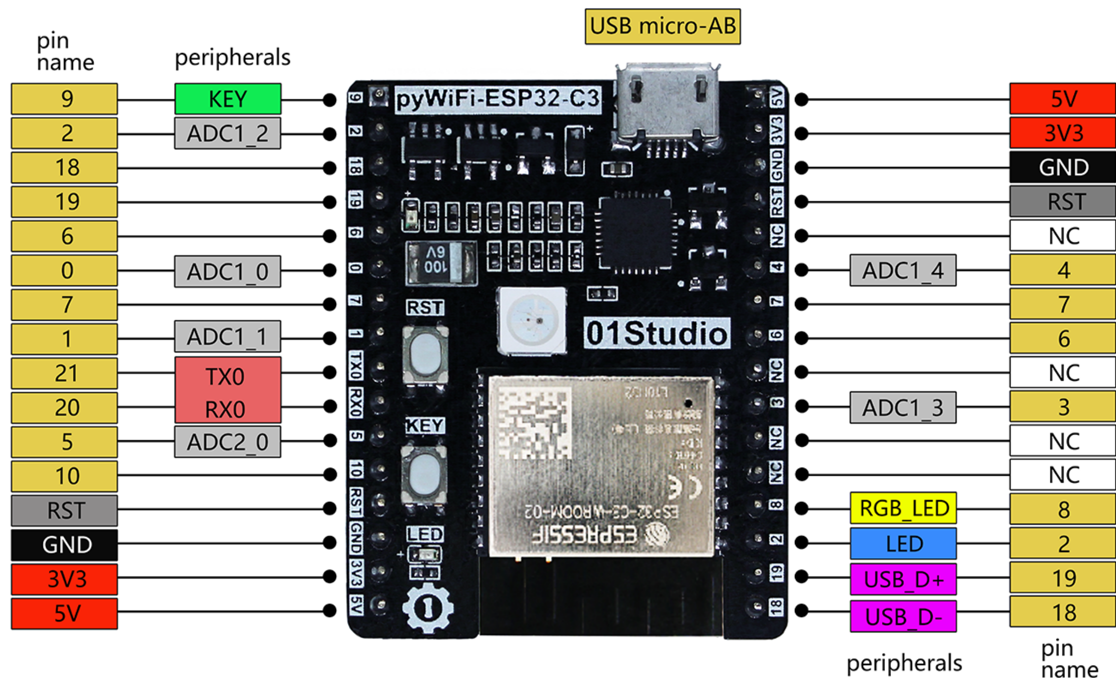
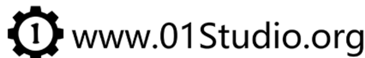
# ESP32-C3 快速参考手册¶




pyWiFi-ESP32-C3

Micro-Python




5V：USB供电输入
3V3：3.3V输出，最大电流600mA
VBAT：锂电池输入（板载FET保护电路和锂电池充电电路，XH-2.54　2P接口在背面）

LED：连接到引脚　'2'
KEY：连接到引脚　'9'
I2C：支持任意IO
UART：支持任意IO
PWM：支持任意IO

ESP32-C3开发板 (图片来源: 01Studio).

## 通用控制¶

MicroPython 的串口交互调试（REPL）在 UART0 (GPIO21=TX, GPIO20=RX)，波特率为：115200。 Tab按键补全功能对于找到每个对象的使用方法非常有用。 粘贴模式 (ctrl-E) 对需要复制比较多 的python代码到REPL是非常有用。

The `machine` module:

```
import machine

machine.freq()          # 获取CPU当前工作频率
machine.freq(240000000) # 设置CPU的工作频率为 240 MHz
```

The `esp` module:

```
import esp

esp.osdebug(None)       # 关闭原厂 O/S 调试信息
```

```
esp.osdebug(0)          # 将原厂 O/S 调试信息重定向到 UART(0) 输出

# 与flash交互的低级方法
esp.flash_size()
esp.flash_user_start()
esp.flash_erase(sector_no)
esp.flash_write(byte_offset, buffer)
esp.flash_read(byte_offset, buffer)
```

The `esp32` module:

```
import esp32

esp32.hall_sensor()      # 读取内部霍尔传感器
esp32.raw_temperature() # 读取内部温度传感器，在MCU上，单位：华氏度F
esp32.ULP()              # 使用超低功耗协处理器（ULP）
```

请注意ESP32内部温度读取数值会比实际要高，因为芯片工作时候回发热。 从睡眠状态唤醒后立即读取温度传感器可以最大限度地减少这种影响。

# Networking¶

The `network` module:

```
import network

wlan = network.WLAN(network.STA_IF) # 创建 station 接口
wlan.active(True)        # 激活接口
wlan.scan()              # 扫描允许访问的SSID
wlan.isconnected()       # 检查创建的station是否连已经接到AP
wlan.connect('essid', 'password') # 连接到指定ESSID网络
wlan.config('mac')       # 获取接口的MAC地址
wlan.ifconfig()          # 获取接口的 IP/netmask(子网掩码)/gw(网关)/DNS 地址

ap = network.WLAN(network.AP_IF) # 创捷一个AP热点接口
ap.config(essid='ESP-AP') # 激活接口
ap.config(max_clients=10) # 设置热点允许连接数量
ap.active(True)          # 设置AP的ESSID名称
```

连接到本地WIFI网络的函数参考:

```
def do_connect():
    import network
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    if not wlan.isconnected():
        print('connecting to network...')
        wlan.connect('essid', 'password')
        while not wlan.isconnected():
            pass
```

```
    print('network config:', wlan.ifconfig())
```

一旦网络建立成功，你就可以通过 `socket` 模块创建和使用 TCP/UDP sockets 通讯, 以及通过 `urequests` 模块非常方便地发送 HTTP 请求。

# 延时和时间¶

Use the `time` module:

```
import time

time.sleep(1)           # 睡眠1秒
time.sleep_ms(500)      # 睡眠500毫秒
time.sleep_us(10)       # 睡眠10微妙
start = time.ticks_ms() # 获取毫秒计时器开始值
delta = time.ticks_diff(time.ticks_ms(), start) # 计算从开始到当前时间的差值
```

# 定时器¶

ESP32-C3拥有2个定时器。使用 machine.Timer 类通过设置timer ID号为 0-1

```
from machine import Timer

tim0 = Timer(0)
tim0.init(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(0))

tim1 = Timer(1)
tim1.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(1))
```

该周期的单位为毫秒(ms)

Virtual timers are not currently supported on this port.

# 引脚和GPIO口¶

使用 machine.Pin 模块:

```
from machine import Pin

p0 = Pin(0, Pin.OUT)    # 创建对象p0，对应GPIO0口输出
p0.on()                 # 设置引脚为 "on" (1)高电平
p0.off()                # 设置引脚为 "off" (0)低电平
p0.value(1)             # 设置引脚为 "on" (1)高电平

p2 = Pin(2, Pin.IN)     # 创建对象p2，对应GPIO2口输入
print(p2.value())       # 获取引脚输入值, 0 (低电平) 或者 1 (高电平)
```

```
p4 = Pin(4, Pin.IN, Pin.PULL_UP) # 打开内部上拉电阻
p5 = Pin(5, Pin.OUT, value=1) # 初始化时候设置引脚的值为 1（高电平）
```

可以使用引脚排列如下 (包括首尾): 0-10, 18-21.分别对应ESP32-C3芯片的实际引脚编号。 请注意，用户使用自己其它的开发板有特定的引脚命名方式（例如：DO, D1, ...）。 由于MicroPython致力于支持不同的开发板和模块，因此我们采用最原始简单具且有共同特征的引 脚命名方式。如果你使用自己的开发板，请参考其原理图。

注意:

- 引脚21和20分别是串口交互（REPL）的TX和RX。
- 引脚19和18分是USB的D+和D-。
- 部分引脚的pull值可以设置为 `Pin.PULL_HOLD` 以降低深度睡眠时候的功耗。

# UART (serial bus)¶

See machine.UART.

```
from machine import UART

uart1 = UART(1, baudrate=115200, tx=6, rx=7)
uart1.write('hello')  # write 5 bytes
uart1.read(5)         # read up to 5 bytes
```

The ESP32-C3 has 2 hardware UARTs: UART0, UART1. They each have default GPIO assigned to them, however depending on your ESP32-C3 variant and board, these pins may conflict with embedded flash, onboard PSRAM or peripherals.

Any GPIO can be used for hardware UARTs using the GPIO matrix, so to avoid conflicts simply provide `tx` and `rx` pins when constructing. The default pins listed below.

| | UART0 | UART1 |
|---|---|---|
| tx | 21 | 任意IO |
| rx | 20 | 任意IO |

# PWM (脉宽调制)¶

There’s a higher-level abstraction machine.Signal which can be used to invert a pin. Useful for illuminating active-low LEDs using `on()` or `value(1)`.

PWM 能在所有可输出引脚上实现。基频的范围可以从 40Hz 到 40MHz 但需要权衡: 随着基频的 增加 占空分辨率 下降. 详情请参阅： LED Control . 现在占空比范围为 0-1023

Use the `machine.PWM` class:

```
from machine import Pin, PWM

pwm0 = PWM(Pin(0))        # 从1个引脚中创建PWM对象
pwm0.freq()               # 获取当前频率
pwm0.freq(1000)           # 设置频率
pwm0.duty()               # 获取当前占空比
pwm0.duty(200)            # 设置占空比
pwm0.deinit()             # 关闭引脚的 PWM

pwm2 = PWM(Pin(2), freq=20000, duty=512) # 在同一语句下创建和配置 PWM
```

# ADC (模数转换)¶

ADC功能在ESP32-C3引脚0-4上可用(ADC1的5个输入通道)。请注意，使用默认配置时，ADC引脚 上的输入电压必须介于0.0v和1.0v之间（任何高于1.0v的值都将读为4095）。如果需要增加测 量范围，需要配置衰减器。

Use the machine.ADC class:

```
from machine import ADC

adc = ADC(Pin(0))           # 在ADC引脚上创建ADC对象
adc.read()                   # 读取测量值, 0-4095 表示电压从 0.0v - 1.0v

adc.atten(ADC.ATTN_11DB)     # 设置 11dB 衰减输入 (测量电压大致从 0.0v - 3.6v)
adc.width(ADC.WIDTH_9BIT)    # 设置 9位 精度输出 (返回值 0-511)
adc.read()                   # 获取重新配置后的测量值
```

ESP32-C3 特定的 ADC 类使用方法说明:

### `ADC.atten(`*attenuation*`)`¶

该方法允许设置ADC输入的衰减量，以获取更大的电压测量范围，但是以精度为代价的。（配置后相同的位数表示更宽的范围）。衰减选项如下:

- `ADC.ATTN_0DB`: 0dB 衰减, 最大输入电压为 1.00v - 这是默认配置
- `ADC.ATTN_2_5DB`: 2.5dB 衰减, 最大输入电压约为 1.34v
- `ADC.ATTN_6DB`: 6dB 衰减, 最大输入电压约为 2.00v

- `ADC.ATTN_11DB`: 11dB 衰减, 最大输入电压约为3v

**Warning**

尽管通过配置11dB衰减可以让测量电压到达3.6v,但由于ESP32-C3芯片的最大允许输入电压是3.6V, 因此输入接近3.6V的电压可能会导致IC烧坏!

### ADC.width(*width*)¶

该方法允许设置ADC输入的位数精度。 选项如下:

- `ADC.WIDTH_9BIT`: 9 bit data
- `ADC.WIDTH_10BIT`: 10 bit data
- `ADC.WIDTH_11BIT`: 11 bit data
- `ADC.WIDTH_12BIT`: 12 bit data - 这是默认配置

# 软件SPI总线¶

EPS32内部有两个SPI驱动。其中1个是通过软件实现 (bit-banging)，并允许配置到所有引脚， 通过 machine.SoftSPI 类模块配置:

```
from machine import Pin, SoftSPI

# 在给定的引脚上创建SoftSPI总线
# （极性）polarity是指 SCK 空闲时候的状态
# （相位）phase=0 表示SCK在第1个边沿开始取样，phase=1 表示在第2个边沿开始。
spi = SoftSPI(baudrate=100000, polarity=1, phase=0, sck=Pin(0), mosi=Pin(2), miso=Pin(4))


spi.init(baudrate=200000) # 设置频率

spi.read(10)            # 在MISO引脚读取10字节数据
spi.read(10, 0xff)      # 在MISO引脚读取10字节数据同时在MOSI输出0xff

buf = bytearray(50)     # 建立缓冲区
spi.readinto(buf)       # 读取数据并存放在缓冲区 (这里读取50个字节)
spi.readinto(buf, 0xff) # 读取数据并存放在缓冲区，同时在MOSI输出0xff

spi.write(b'12345')     # 在MOSI引脚上写5字节数据

buf = bytearray(4)      # 建立缓冲区
spi.write_readinto(b'1234', buf) # 在MOSI引脚上写数据并将MISO读取数据存放到缓冲区
spi.write_readinto(buf, buf) # 在MOSI引脚上写缓冲区的数据并将MISO读取数据存放到缓冲区
```

**Warning**

目前在创建软件SPI对象时，`sck`, `mosi` 和 `miso` 所有 的引脚 必须 定义。

## 硬件SPI总线¶

有两个硬件SPI通道允许更高速率传输（到达80MHz）。 也可以配置成任意引脚，但相关引脚要 符合输入输出的方向性，这可以参阅(see 引脚和GPIO口)内容。通过自定义引脚而非 默认引脚，会降低传输速度，上限为40MHz。以下是硬件SPI总线默认引脚：

| | HSPI (id=1) | VSPI (id=2) |
|---|---|---|
| sck | 14 | 18 |
| mosi | 13 | 23 |
| miso | 12 | 19 |

Hardware SPI is accessed via the machine.SPI class and has the same methods as software SPI above:

```
from machine import Pin, SPI

hspi = SPI(1, 10000000)
hspi = SPI(1, 10000000, sck=Pin(14), mosi=Pin(13), miso=Pin(12))
vspi = SPI(2, baudrate=80000000, polarity=0, phase=0, bits=8,
firstbit=0, sck=Pin(18), mosi=Pin(23), miso=Pin(19))
```

## SoftI2C总线¶

I2C总线分软件和硬件对象，硬件可以定义0和1，通过配置可以在任意引脚上实现改功能， 详情请看 machine.SoftI2C 类模块:

```
from machine import Pin, SoftI2C

# 构建1个I2C对象
i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)

# 构建一个硬件 I2C 总线
i2c = I2C(0)
i2c = I2C(1, scl=Pin(5), sda=Pin(4), freq=400000)

i2c.scan()              # 扫描从设备

i2c.readfrom(0x3a, 4)   # 从地址为0x3a的从机设备读取4字节数据
i2c.writeto(0x3a, '12') # 向地址为0x3a的从机设备写入数据"12"
```

```
buf = bytearray(10)      # 创建1个10字节缓冲区
i2c.writeto(0x3a, buf)  # 写入缓冲区数据到从机
```

# Hardware I2C bus

There are two hardware I2C peripherals with identifiers 0 and 1. Any available output-capable pins can be used for SCL and SDA but the defaults are given below.

| | I2C(0) | I2C(1) |
|---|---|---|
| scl | 18 | 25 |
| sda | 19 | 26 |

The driver is accessed via the machine.I2C class and has the same methods as software I2C above:

```
from machine import Pin, I2C

i2c = I2C(0)
i2c = I2C(1, scl=Pin(5), sda=Pin(4), freq=400000)
```

# 实时时钟(RTC)

See machine.RTC

```
from machine import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 1, 12, 48, 0, 0)) # 设置时间（年，月，日，星期，时，分，秒，微秒）
                                             # 其中星期使用0-6表示星期一至星期日。
rtc.datetime() # 获取当前日期和时间
```

# WDT (Watchdog timer)

See machine.WDT.

```
from machine import WDT

# enable the WDT with a timeout of 5s (1s is the minimum)
wdt = WDT(timeout=5000)
wdt.feed()
```

# 深度睡眠模式

下面代码可以用来睡眠、唤醒和检测复位唤醒:

```
import machine
```

```python
# 检测设备是否从深度睡眠中唤醒
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
    print('woke from a deep sleep')

# 使设备进入深度睡眠，时间10秒。
machine.deepsleep(10000)
```

注意事项:

- 调用深度睡眠函数 `deepsleep()` 如果不提供参数（时间）的话可能会让设备无限期休眠。
- 软件复位不能触发复位事件（reset cause）。
- 可能会出现一些泄漏电流流经内部上下拉电阻，为了进一步降低功耗，可以关闭GPIO的上下拉电阻:
  ```python
  p1 = Pin(4, Pin.IN, Pin.PULL_HOLD)
  ```
- 退出深度睡眠后，有必要恢复GPIO原来的状态 (例如：原来是输出引脚)
  ```python
  p1 = Pin(4, Pin.OUT, None)
  ```

# RMT¶

The RMT is ESP32-specific and allows generation of accurate digital pulses with 12.5ns resolution. See esp32.RMT for details. Usage is:

```python
import esp32
from machine import Pin

r = esp32.RMT(0, pin=Pin(18), clock_div=8)
r   # RMT(channel=0, pin=18, source_freq=80000000, clock_div=8)
# The channel resolution is 100ns (1/(source_freq/clock_div)).
r.write_pulses((1, 20, 2, 40), start=0) # Send 0 for 100ns, 1 for 2000ns, 0 for 200ns, 1 for 4000ns
```

# 单总线驱动（Onewire）¶

单总线驱动允许通过软件在各个引脚上实现:

```python
from machine import Pin
import onewire

ow = onewire.OneWire(Pin(12)) # 在引脚 GPIO12 创建单总线对象ow
```

```
ow.scan()               # 扫描设备,返回设备编号列表
ow.reset()              # 复位总线
ow.readbyte()           # 读取1字节
ow.writebyte(0x12)      # 写入1个字节（0x12）
ow.write('123')         # 写入多个字节('123')
ow.select_rom(b'12345678') # 根据ROM编号选择总线上的指定设备
```

下面是一个DS18B20设备的驱动函数:

```
import time, ds18x20
ds = ds18x20.DS18X20(ow)
roms = ds.scan()
ds.convert_temp()
time.sleep_ms(750)
for rom in roms:
    print(ds.read_temp(rom))
```

确保数据引脚连接了 4.7k 的上拉电阻。另外请注意每次采集温度都需要用到 `convert_temp()` 模块。

# NeoPixel and APA106 driver¶

Use the `neopixel` and `apa106` modules:

```
from machine import Pin
from neopixel import NeoPixel

pin = Pin(0, Pin.OUT)   # 设置引脚GPIO0来驱动 NeoPixels
np = NeoPixel(pin, 8)   # 在GPIO0上创建一个 NeoPixel对象，包含8个灯珠
np[0] = (255, 255, 255) # 设置第一个灯珠显示数据为白色
np.write()              # 写入数据
r, g, b = np[0]         # 获取第一个灯珠的颜色
```

The APA106 driver extends NeoPixel, but internally uses a different colour order:

```
from apa106 import APA106
ap = APA106(pin, 8)
r, g, b = ap[0]
```

APA102 (DotStar) uses a different driver as it has an additional clock pin.

低级别的 NeoPixel 驱动:

```
import esp
esp.neopixel_write(pin, grb_buf, is800khz)
```

**Warning**

默认情况下 `NeoPixel` 被配置成控制更常用的 *800kHz* 单元设备。用户可以通过使用替代的定时器 来说控制其他频率的设备 (通常是 400kHz)。 可以通过使用定时器 `timing=0` 当构建 `NeoPixel` 对象的时候。

# DHT 驱动¶

DHT 温湿度驱动允许通过软件在各个引脚上实现:

```
import dht
import machine

d = dht.DHT11(machine.Pin(4))
d.measure()
d.temperature() # eg. 23 (°C)
d.humidity()    # eg. 41 (% RH)

d = dht.DHT22(machine.Pin(4))
d.measure()
d.temperature() # eg. 23.6 (°C)
d.humidity()    # eg. 41.3 (% RH)
```

# WebREPL (Web浏览器交互提示)¶

WebREPL (通过WebSockets的REPL, 可以通过浏览器使用) 是ESP8266端口实验的功能。 可以从 https://github.com/micropython/webrepl 下载并打开html文件运行。 (在线托管版可以通过访问 http://micropython.org/webrepl)直接使用, 通过执行 以下命令进行配置:

```
import webrepl_setup
```

按照屏幕的提示操作。重启后，允许使用WebREPL。如果你禁用了开机自动启动WebREPL, 可以通过以下命令使用:

```
import webrepl
webrepl.start()

# 也可在在开始时候设置一个密码
webrepl.start(password='mypass')
```

这个 WebREPL 通过连接到ESP32-C3的AP使用,如果你的路由器配网络配置正确，这个功能 也可以通过STA方式使用，那意味着你可以同时上网和调试ESP32-C3。(如果遇到不可行的 特殊情况， 请先使用ESP32-C3 AP方式)。

除了终端/命令符的访问方式, WebREPL同时允许传输文件 (包含上传和下载)。Web客户端有相应的 功能按钮，也可以通过 `webrepl_cli.py` 模块上存储的命令行进行操作。

有关将文件传输到ESP32-C3其他支持的替代方法，请参阅MicroPython论坛