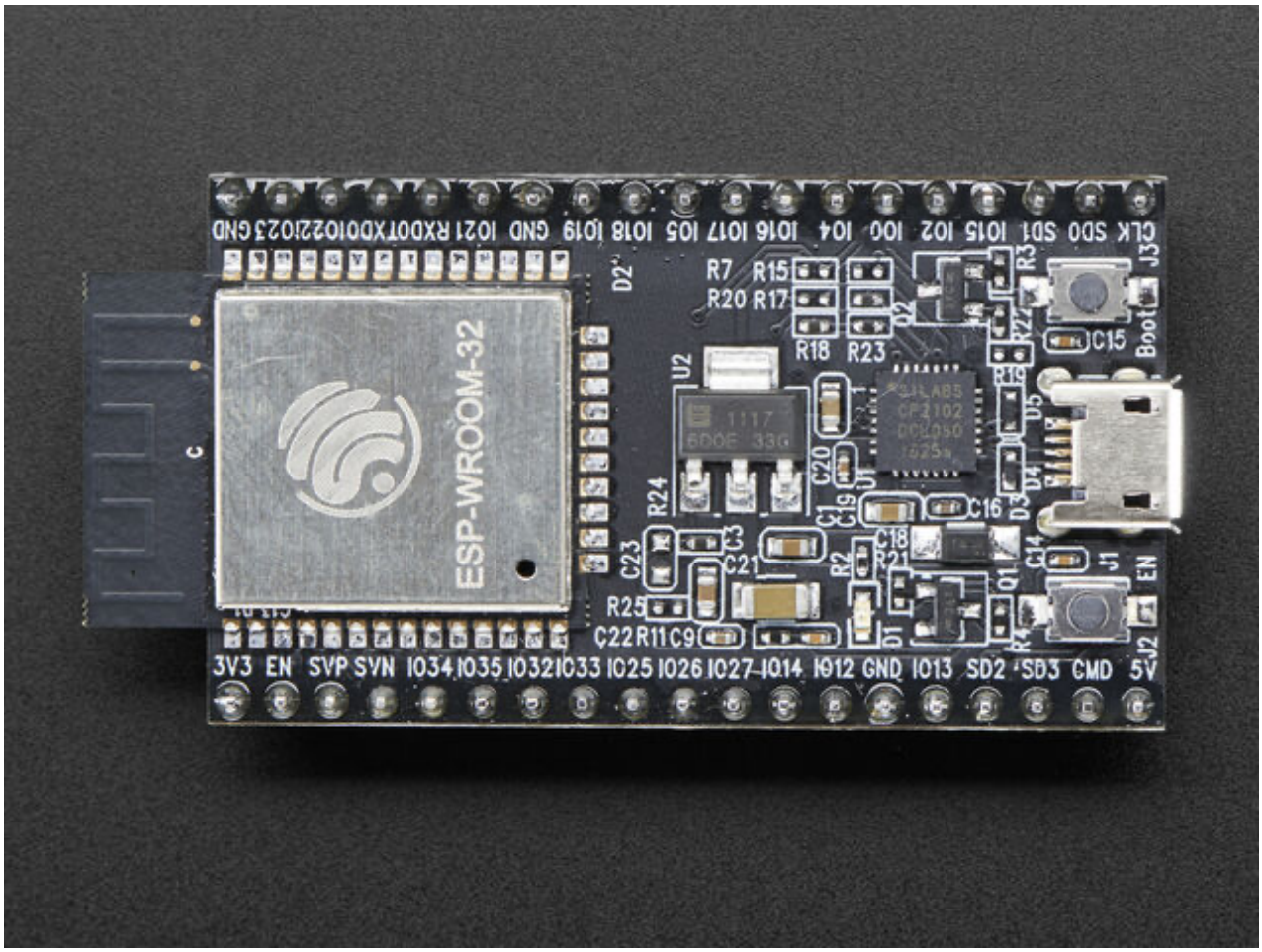
このドキュメンテーションは、MicroPython の最新開発ブランチのためのものです。 リリースバージョンでは利用できない機能に言及することがあります。

特定のリリースのドキュメントをお探しの場合は、左側のドロップダウンメニューを使って、 望みのバージョンを選択します。

ESP32 用クイックリファレンス



Espressif ESP32 Development Board (画像出所: Adafruit)

以下は、ESP32 ベースのボードのためのクイックリファレンスです。このボードを初めて使う場合は、まず次のマイクロコントローラの概要を確認してみてください。

- ESP32 ポートに関する一般的なこと
- ESP32用 MicroPython チュートリアル

MicroPython のインストール

チュートリアルの章: ESP32 での MicroPython の始め方 を参照してください。そこにはトラブルシューティングについても記載されています。

ボードの一般的な制御

MicroPython REPL は、ボーレート 115200 の UART0 (GPIO1 = TX、GPIO3 = RX) で利用できます。タブ補完は、オブジェクトにどのようなメソッドがあるかを調べるのに便利です。貼り付けモード(ctrl-E)は、大きめの Pythonコードを REPL に貼り付けるのに便利です。

`machine` モジュール:

```
import machine

machine.freq()          # CPU の現在の周波数を取得
machine.freq(240000000) # CPU の周波数を 240 MHz に設定
```

`esp` モジュール:

```
import esp

esp.osdebug(None)       # ベンダ O/S デバッグメッセージをオフにする
esp.osdebug(0)          # ベンダ O/S デバッグメッセージを UART(0) にリダイレクト

# フラッシュストレージを操作するための低レベルのメソッド
esp.flash_size()
esp.flash_user_start()
esp.flash_erase(sector_no)
esp.flash_write(byte_offset, buffer)
esp.flash_read(byte_offset, buffer)
```

`esp32` モジュール:

```
import esp32

esp32.hall_sensor()      # 内部ホールセンサーを読む
esp32.raw_temperature() # MCUの内部温度を華氏で読む
esp32.ULP()              # 超低消費電力コプロセッサへのアクセス
```

ESP32 の温度センサは、動作中に IC が暖かくなるため、通常は周囲温度より高くなります。この影響は、スリープから起床した直後に温度センサーを読むことによって最小限に抑えられます。

ネットワーキング

WLAN

`network` モジュール:

```
import network

wlan = network.WLAN(network.STA_IF) # ステーションインタフェースを作成
wlan.active(True)       # インタフェースをアクティブ化
wlan.scan()             # アクセスポイントをスキャン
wlan.isconnected()      # ステーションが AP に繋がったかをチェック
wlan.connect('ssid', 'key') # AP に接続
wlan.config('mac')      # インタフェースの MAC アドレスを取得
wlan.ifconfig()         # インタフェースの IP/netmask/gw/DNS アドレスを取得

ap = network.WLAN(network.AP_IF) # アクセスポイントインタフェースを作成
ap.config(ssid='ESP-AP') # アクセスポイントの SSID を設定
ap.config(max_clients=10) # ネットワークに接続できるクライアント数を設定
ap.active(True)         # インタフェースをアクティブ化
```

ローカルの WiFi ネットワークに接続するには、次の関数を流用してください:

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

ネットワークが確立されると `socket` モジュールを使って、通常どおり TCP/UDP ソケットを作成して使用できます。HTTP リクエストするには `urequests` モジュールを使うと便利です。

`wlan.connect()` を呼んだ後、認証に失敗した場合やAPが範囲内にない場合でも、デバイスはデフォルトで **永遠** に接続を再試行します。 この状態で `wlan.status()` は、接続が成功するかインターフェースが無効になるまで、`network.STAT_CONNECTING` を返します。これは `wlan.config(reconnects=n)` を呼び出すことで変更できます。n は再接続の回数です(0 は再試行しないことを意味し、-1 は永遠に再接続を試みるデフォルトの動作になります)。

LAN

有線インターフェースを利用するには、ピンとモードを指定する必要があります

```
import network

lan = network.LAN(mdc=PIN_MDC, ...)   # ピンとモードを設定
lan.active(True)                      # インタフェースをアクティブ化
lan.ifconfig()                        # インタフェースのIP/ネットマスク/gw/DNS アドレスを取得
```

PHY タイプとインタフェースを定義するコンストラクタのキーワード引数は次のとおりです:

- mdc=pin-object # mdc と mdio ピンを設定。
- mdio=pin-object
- power=pin-object # PHY デバイスの電源を切り換えるピンを設定。
- phy_type=<type> # PHYデバイスの種類を選択。サポートされているデバイスは、PHY_LAN8710, PHY_LAN8720, PH_IP101, PHY_RTL8201, PHY_DP83848,

PHY_KSZ8041。
- phy_addr=number # PHY デバイスのアドレス番号。
- ref_clk_mode=mode # ESP32 の ref_clk が入力であるか出力であるかを定義。適切な値は Pin.IN と Pin.OUT。
- ref_clk=pin-object # ref_clkに使用するピンを定義。

ref_clk_mode と ref_clk のオプションは、少なくとも esp-idf バージョン 4.4 が必要です。それ以前のバージョンの esp-idf では、これらのパラメータは kconfig ボードオプションで定義しなければなりません。

次のコードは、一般的なボードの LAN インタフェースで動作する設定です:

```
# Olimex ESP32-GATEWAY: Pin(5) で電源制御
# Olimex ESP32 PoE と ESP32-PoE ISO: Pin(12) で電源制御

lan = network.LAN(mdc=machine.Pin(23), mdio=machine.Pin(18),
power=machine.Pin(5),
                  phy_type=network.PHY_LAN8720, phy_addr=0)

# ダイナミック ref_clk ピン設定のある場合

lan = network.LAN(mdc=machine.Pin(23), mdio=machine.Pin(18),
power=machine.Pin(5),
                  phy_type=network.PHY_LAN8720, phy_addr=0,
                  ref_clk=machine.Pin(17), ref_clk_mode=machine.Pin.OUT)

# Wireless-Tag WT32-ETH01 の場合

lan = network.LAN(mdc=machine.Pin(23), mdio=machine.Pin(18),
                  phy_type=network.PHY_LAN8720, phy_addr=1, power=None)

# Espressif ESP32-Ethernet-Kit_A_V1.2 の場合

lan = network.LAN(id=0, mdc=Pin(23), mdio=Pin(18), power=Pin(5),
                  phy_type=network.PHY_IP101, phy_addr=1)
```

sdkconfig.board ファイルにおける PHY インタフェースの適切な定義は次のとおりです:

```
CONFIG_ETH_PHY_INTERFACE_RMII=y
CONFIG_ETH_RMII_CLK_OUTPUT=y
CONFIG_ETH_RMII_CLK_OUT_GPIO=17
CONFIG_LWIP_LOCAL_HOSTNAME="ESP32_POE"
```

CONFIG_ETH_RMII_CLK_OUT_GPIO に割り当てられた値は、ボードの配線によっ

て異なる場合があります。

遅延とタイミング

`time` モジュールを使います:

```
import time

time.sleep(1)           # 1秒間、一時停止する
time.sleep_ms(500)      # 500ミリ秒間、一時停止する
time.sleep_us(10)       # 10マイクロ秒間、一時停止する
start = time.ticks_ms() # ミリ秒カウンター値を取得
delta = time.ticks_diff(time.ticks_ms(), start) # 時差を計算
```

タイマー

ESP32 には４つのハードウェアタイマーがあります。 machine.Timer クラスに 0 から 3 までのタイマーIDを指定して使います。

```
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))
```

period の単位はミリ秒です。

仮想タイマーは、このポートではサポートしていません。

ピンと GPIO

machine.Pin クラスを使います:

```
from machine import Pin

p0 = Pin(0, Pin.OUT)    # create output pin on GPIO0
p0.on()                 # set pin to "on" (high) level
p0.off()                # set pin to "off" (low) level
p0.value(1)             # set pin to on/high

p2 = Pin(2, Pin.IN)     # create input pin on GPIO2
print(p2.value())       # get value, 0 or 1

p4 = Pin(4, Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor
p5 = Pin(5, Pin.OUT, value=1) # set pin high on creation
p6 = Pin(6, Pin.OUT, drive=Pin.DRIVE_3) # set maximum drive strength
```

使用可能なピンは、ESP32 チップの実際の GPIO ピン番号に対応する 0-19, 21-23, 25-27, 32-39 です。これらは ESP32 チップの実際の GPIO ピン番号に対応しています。多くのエンドユーザーボードでは、独自のアドホックピン番号(D0、D1、... など)が使われています。ボードの論理ピンと物理的なチップピンとのマッピングについては、ボードのマニュアルを参照してください。

４つのドライブ電流値(drive strength)がサポートされています。これは `Pin()` コンストラクタまたは `Pin.init()` メソッドの `drive` キーワード引数で設定できます。４つのドライブ電流値は、それぞれ異なる安全最大ソース/シンク電流とおおよその内部ドライバー抵抗を持ちます。

- `Pin.DRIVE_0`: 5mA / 130 Ω
- `Pin.DRIVE_1`: 10mA / 60 Ω
- `Pin.DRIVE_2`: 20mA / 30 Ω (設定しない場合のデフォルトのドライブ電流値)
- `Pin.DRIVE_3`: 40mA / 15 Ω

`Pin()` と `Pin.init()` の `hold=` キーワード引数は、ESP32の「パッドホールド」(pad hold)機能を有効にします。 `True` に設定すると、ピンの構成(入力/出力の方向、プル抵抗、出力値)が保持され、それ以降の変更は適用されません(出力レベルの変更を含む)。 `hold=False` に設定すると、未変更のピン設定変更が直ちに適用され、ピンが解放されます。ピンがすでにホールドされている状態で `hold=True` に設定すると、設定変更を適用し、その直後にホールドが再適用されます。

注記:

- ピン 1 と 3 は、それぞれ REPL UART の TX と RX

- ピン 6, 7, 8, 11, 16, 17 は内蔵フラッシュの接続に使っているので、他の目的で使うのは推奨しません
- ピン 34-39 は入力専用で、内部プルアップ抵抗もありません
- スリープ時のピンの挙動については ディープスリープモード を参照してください。

ピンを反転させるのに使える、高レベルの抽象化インタフェース machine.Signal があります。 負論理(active-low)の LED でも `on()` や `value(1)` で点灯できるので便利です。

UART (シリアルバス)

machine.UART を参照

```
from machine import UART

uart1 = UART(1, baudrate=9600, tx=33, rx=32)
uart1.write('hello')  # 5バイト書き出す
uart1.read(5)         # 5バイトまで読み込む
```

ESP32 には３つのハードウェア UART があります: UART0, UART1, UART2 です。それぞれにデフォルトの GPIO が割り当てられていますが、ESP32 の種類やボードによっては、これらのピンが内蔵フラッシュ、オンボードの PSRAM、ペリフェラルと競合しているかもしれません。

GPIO マトリクスを使えば、どの GPIO もハードウェア UART に使えます。例外はピン 34-39 で、これらは入力専用なので `rx` にしか使えません。競合を回避するには、コンストラクト時に tx と rx ピンを指定するだけです。デフォルトのピンは以下の通りです。

	UART0	UART1	UART2
tx	1	10	17
rx	3	9	16

PWM (パルス幅変調)

PWM はすべての出力対応ピンで有効にできます。基本周波数は 1Hz から 40MHz の範囲ですが、トレードオフがあります。ベース周波数が高くなると、デューティ分解能は低下します詳細については LED制御 を参照してください。

machine.PWM クラスを使います:

```
from machine import Pin, PWM

pwm0 = PWM(Pin(0))          # ピンから PWM オブジェクトを作成
freq = pwm0.freq()          # 現在の周波数を取得(デフォルト 5kHz)
pwm0.freq(1000)             # 1Hz から 40MHz の範囲で PWM 周波数を設定

duty = pwm0.duty()          # 現在のデューティ比(0-1023 の範囲)を取得(デフォルト 512、50%)
pwm0.duty(256)              # 0 から 1023 の範囲でデューティ比(duty/1023)を設定(この場合は 25%)

duty_u16 = pwm0.duty_u16() # 現在のデューティ比(0-65535 の範囲)を取得
pwm0.duty_u16(2**16*3//4)  # 0 から 65535 の範囲でデューティ比(duty_u16/65525)を設定(この場合は 75%)

duty_ns = pwm0.duty_ns()    # 現在のパルス幅を ns 単位で取得
pwm0.duty_ns(250_000)       # ナノ秒でパルス幅を 0 から 1_000_000_000/freq の範囲で設定(この場合は 25%)

pwm0.deinit()               # このピンの PWM を無効化

pwm2 = PWM(Pin(2), freq=20000, duty=512)  # 作成と設定を一度に実行
print(pwm2)                               # PWM 設定を表示
```

ESP チップによってハードウェアのペリフェラルは異なります:

ハードウェア仕様	ESP32	ESP32-S2	ESP32-C3
グループ(スピードモード)数	2	1	1
グループ毎のタイマー数	4	4	4
グループ毎のチャンネル数	8	8	6
PWM 周波数の種類(グループ数 * タイマー数)	8	4	4
全 PWM チャンネル数(ピン、デューティ) (グループ数 * チャンネル数)	16	8	6

ESP32では最大16個のPWMチャンネル(ピン)を使えますが、利用できるPWM周波数は8種類のみで、残りの8チャンネルは同じ周波数でなければなりません。一方, 同じ周波数であれば, 16の独立したPWMデューティ比が可能です。

さらなる例は パルス幅変調 チュートリアルを参照。

ADC (アナログ/デジタル変換)

ESP32 で ADC 機能はピン 32-39 (ADC ブロック1)とピン 0, 2, 4, 12-15, 25-27 (ADC ブロック2)で利用できます。

machine.ADC クラスを使います:

```
from machine import ADC

adc = ADC(pin)          # 指定のピンについて ADC オブジェクトを作成
val = adc.read_u16()    # 生のアナログ値を 0-65535 の範囲で読込み
val = adc.read_uv()     # アナログ値をマイクロボルト単位で読込み
```

ADC ブロック2は WiFi でも使っているため、WiFi がアクティブなときにブロック2ピンからアナログ値を読み込もうとすると例外が発生します。

内部 ADC の基準電圧は通常 1.1V ですが、パッケージによって若干の違いがあります。ADC は基準電圧の近くでは線形性が悪く(特に高減衰の場合)、100mV 程度の最小測定電圧があり、これ以下の電圧は 0 として読まれます。電圧を正確に読むには `read_uv()` メソッド(下記参照)を使うことが推奨されます。

ESP32 固有の ADC クラスのメソッドリファレンス:

class **ADC**(*pin, *, atten*)

指定したピンの ADC オブジェクトを返します。ESP32 は異なるタイミングの ADC サンプリングをサポートしていないため、 `sample_ns` キーワード引数はサポートしていません。

基準電圧以上の電圧を読み取るには、キーワード引数 `atten` で入力減衰率を適用します。有効な値(およびおおよその線形測定範囲)は以下のとおりです:

- `ADC.ATTN_0DB`: 減衰率を適用しません(100mV - 950mV)
- `ADC.ATTN_2_5DB`: 2.5dB の減衰率(100mV - 1250mV)
- `ADC.ATTN_6DB`: 6dB の減衰率(150mV - 1750mV)
- `ADC.ATTN_11DB`: 11dB の減衰率(150mV - 2450mV)

警告

入力ピンの絶対最大定格電圧は 3.6V であり、この境界線に近づくと IC に損傷を与える可能性があることに注意してください！

ADC.read_uv()

このメソッドは、ADC の既知の特性と、製造時に設定されたパッケージごとの eFuse 値を使って、校正された入力電圧(減衰前)をマイクロボルト単位で返します。返される値はミリボルトの分解能しかありません(つまり、常に 1000 マイクロボルトの倍数です)。

キャリブレーションは、ADC のリニアレンジにおいてのみ有効です。特に、グランドに接続された入力は、0 マイクロボルトを超える値として読み取られます。しかし、線形範囲内では、*read_u16()* を使って結果を定数でスケーリングするよりも、より正確で一貫した結果が得られるでしょう。

ESP32 ポートは machine.ADC API もサポートします:

class **ADCBlock**(*id, *, bits*)

指定した `id` (1 または 2) の ADC ブロックオブジェクトを返し、指定した分解能(ESP32 シリーズによって 9 から 12 ビット)に初期化します。分解能を指定しない場合はサポートされている最高の分解能になります。

ADCBlock.connect(*pin*)

ADCBlock.connect(*channel*)

ADCBlock.connect(*channel, pin*)

指定した ADC ピンまたはチャンネル番号に対応する `ADC` オブジェクトを返します。GPIO への ADC チャンネルの任意の接続はサポートされていないため、このブロックに接続されていないピンを指定したり、チャンネルとピンが不一致の場合は例外が発生します。

Legacy methods:

ADC.read()

このメソッドは ADC ブロックの分解能(デフォルトの12ビット分解能では0〜4095)にしがかった範囲の生の ADC 値を返します。

ADC.atten(*atten*)

`ADC.init(atten=atten)` と同等です。

ADC.width(*bits*)

`ADC.block().init(bits=bits)` と同等です。

互換性のため、ADC オブジェクトはサポートされる ADC 分解能に対応した定数も提供します:

- `ADC.WIDTH_9BIT` = 9
- `ADC.WIDTH_10BIT` = 10
- `ADC.WIDTH_11BIT` = 11
- `ADC.WIDTH_12BIT` = 12

ソフトウェア SPI バス

ソフトウェア SPI (ビットバンギング)はすべてのピンで動作し、 machine.SoftSPI クラスを介してアクセスします:

```
from machine import Pin, SoftSPI

# 与えたピンから SoftSPI バスを構築
# 極性 polarity は SCK のアイドル状態
# phase=0 は SCK の第１エッジでサンプルを意味、chase=1 は第２を意味
spi = SoftSPI(baudrate=100000, polarity=1, phase=0, sck=Pin(0), mosi=Pin(2),
miso=Pin(4))

spi.init(baudrate=200000) # ボーレートを設定

spi.read(10)            # MISO で 10 バイト読込み
spi.read(10, 0xff)      # 10 バイト読込み、その間 MOSI に 0xff を出力

buf = bytearray(50)     # バッファを作成
spi.readinto(buf)       # 与えたバッファに読込み(この場合は 50 バイト)
spi.readinto(buf, 0xff) # 与えたバッファに読込み、MOSI に 0xff を出力

spi.write(b'12345')     # MOSI に 5 バイト書込み

buf = bytearray(4)      # バッファを作成
spi.write_readinto(b'1234', buf) # MOSI に書き込み、MISO からバッファに読み込み
spi.write_readinto(buf, buf) # MOSI に buf を書き込み、MISO から buf に読み込み
```

警告

現在のところ、ソフトウェア SPI を初期化するときには `sck`, `mosi`, `miso` すべてを指定しなければなりません。

ハードウェア SPI バス

より高速な伝送速度を可能にする２つのハードウェア SPI チャネルがあります(最大80MHz)。SPI チャンネルには、必要となる I/O 方向をサポートし、他で使われていないものであればどの I/O ピンでも使えます(ピンと GPIO を参照)。しかし、デフォルトで SPI に設定されていないピンについては、GPIO マルチプレクサの追加層を通す必要があります。これは高速での信頼性に影響を与えます。次のデフォルト以外のピンで使用した場合、ハードウェア SPI チャネルは 40MHz に制限されます。

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

ハードウェア SPI には machine.SPI クラスを使ってアクセスします。このクラスには先述のソフトウェア SPI と同じメソッドがあります:

```
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))
```

ソフトウェア I2C バス

ソフトウェア I2C (ビット・バンギングを使用)は、出力可能なすべてのピンで動作し、 machine.SoftI2C クラスを使ってアクセスします。

```
from machine import Pin, SoftI2C

i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)

i2c.scan()              # デバイスをスキャン

i2c.readfrom(0x3a, 4)   # アドレス 0x3a のデバイスから 4 バイト読み込み
i2c.writeto(0x3a, '12') # アドレス 0x3a のデバイスに '12' を書き込み

buf = bytearray(10)     # 10バイトのバッファを作成
i2c.writeto(0x3a, buf)  # 与えたバッファをペリフェラルに書き込み
```

ハードウェア I2C バス

２つのハードウェア I2C ペリフェラルがあり、識別子 0 と 1 がついています。利用可能な出力対応ピンはすべて SCL と SDA にできますが、デフォルトは以下のようになっています。

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

ハードウェア I2C には machine.I2C クラスを使ってアクセスしますこのクラスには先述のソフトウェア I2C と同じメソッドがあります:

```
from machine import Pin, I2C

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

I2S バス

machine.I2S を参照。

```
from machine import I2S, Pin

i2s = I2S(0, sck=Pin(13), ws=Pin(14), sd=Pin(34), mode=I2S.TX, bits=16,
format=I2S.STEREO, rate=44100, ibuf=40000) # create I2S object
i2s.write(buf)             # オーディオサンプルのバッファを I2S デバイスに書き出す

i2s = I2S(1, sck=Pin(33), ws=Pin(25), sd=Pin(32), mode=I2S.RX, bits=16,
format=I2S.MONO, rate=22050, ibuf=40000) # create I2S object
i2s.readinto(buf)          # I2S デバイスからのオーディオサンプルをバッファに読み込む
```

I2S クラスは現在、テクニカルプレビューとして利用できます。プレビュー期間中は、ユーザーからのフィードバックを歓迎します。このフィードバックに基づいて、I2S クラス API と実装が変更される可能性があります。

ESP32には id=0 と id=1 の2つの I2S バスがあります。

リアルタイムクロック (RTC)

machine.RTC を参照:

```
from machine import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 1, 12, 48, 0, 0)) # 指定の日時を設定
rtc.datetime() # 日時を取得
```

WDT (ウォッチドッグタイマー)

machine.WDT を参照:

```
from machine import WDT

# WDT を有効化し、タイムアウトを 5s に設定(最低値は 1s)
wdt = WDT(timeout=5000)
wdt.feed()
```

ディープスリープモード

次のコードで、スリープ、起床、リセット原因のチェックが行えます:

```
import machine

# ディープスリープから起こされたかをチェック
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
    print('woke from a deep sleep')

# 10秒間のディープスリープに入る
machine.deepsleep(10000)
```

注記:

- `deepsleep()` を引数なしで呼び出すと、デバイスは無期限にスリープします
- ソフトウェアリセットによってリセットの原因が変わることはありません

一部の ESP32 ピン(0、2、4、12-15、25-27、32-39)はディープスリープ中に RTC に接続され、 `esp32` モジュールの `wake_on_` 関数でデバイスを起動するのに使えます。出力可能な RTC ピン(34-39 を除くすべて)は、ディープスリープに入るときにもプルアップまたはプルダウン抵抗の構成を保持します。

ディープスリープ中にプル抵抗が積極的に必要とされず、電流リークの原因となる可能性がある場合(例えば、プルアップ抵抗がスイッチを介してグランドに接続されている場合)、ディープスリープモードに入る前に電力を節約するためにそれらを無効にする必要があります。

```
from machine import Pin, deepsleep

# ブート時は入力 RTC ピンをプルアップ付きに設定
pin = Pin(2, Pin.IN, Pin.PULL_UP)

# プルアップを無効にして10秒間スリープさせる
pin.init(pull=None)
machine.deepsleep(10000)
```

`Pin.init()` の `hold=True` 引数でパッドホールドを有効にすると、出力設定された RTC ピンもディープスリープ時にその出力方向とレベルを保持するようになります。

RTC 以外の GPIO ピンは、ディープスリープに入るとデフォルトで接続が解除されます。ピンのパッドホールドを有効にし、ディープスリープ中に GPIO パッドホールドを有効にすると、出力レベルを含む RTC 以外のピンの設定を保持できます。

```
from machine import Pin, deepsleep
import esp32

opin = Pin(19, Pin.OUT, value=1, hold=True) # hold output level
ipin = Pin(21, Pin.IN, Pin.PULL_UP, hold=True) # hold pull-up

# RTC 以外の GPIO のディープスリープ時のパッドホールドを有効にする
esp32.gpio_deep_sleep_hold(True)

# 10秒スリープさせる
deepsleep(10000)
```

スリープからの復帰時には、パッドホールドを含むピンコンフィギュレーションが保持されます。パッドホールドの詳細については、上記の ピンと GPIO を参照してください。

SD カード

machine.SDCard を参照:

```
import machine, os

# スロット 2 はピン sck=18, cs=5, miso=19, mosi=23 を使用
sd = machine.SDCard(slot=2)
os.mount(sd, '/sd')  # マウント

os.listdir('/sd')    # ディレクトリ内容の一覧

os.umount('/sd')     # 取出し
```

RMT

RMT は ESP32 固有であり、12.5ns の分解能で正確なデジタルパルスを生成できます。詳細については esp32.RMT を参照してください。使い方は次のとおりです。

```
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)
# チャンネルの分解能は 100ns (1/(source_freq/clock_div)).
r.write_pulses((1, 20, 2, 40), 0)  # 0 を 100ns, 1 を 2000ns, 0 を 200ns, 1 を
4000ns 送信
```

OneWire ドライバー

OneWire ドライバーはソフトウェアで実装され、すべてのピンで動作します:

```
from machine import Pin
import onewire

ow = onewire.OneWire(Pin(12)) # GPIO 12 で OneWire バスを作成
ow.scan()               # バス上のデバイスリストをスキャン
ow.reset()              # バスをリセット
ow.readbyte()           # 1バイト読込み
ow.writebyte(0x12)      # バスに1バイト書込み
ow.write('123')         # バスに複数バイト書込み
ow.select_rom(b'12345678') # ROM コードで指定したデバイスを選択
```

DS18S20 と DS18B20 のデバイスに対応したドライバが用意されています:

```
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))
```

4.7k のプルアップ抵抗をデータラインに接続してください。 `convert_temp()` メソッドは、温度をサンプリングするたびに呼び出す必要があることに注意してください。

NeoPixel/APA106 ドライバー

`neopixel` と `apa106` モジュールを使います:

```
from machine import Pin
from neopixel import NeoPixel

pin = Pin(0, Pin.OUT)   # NeoPixel 駆動のための GPIO 0 を出力に設定
np = NeoPixel(pin, 8)   # 8ピクセル用の NeoPixel ドライバーを GPIO 0 で作成
np[0] = (255, 255, 255) # 第1ピクセルを白に設定
np.write()              # 全ピクセルにデータ書込み
r, g, b = np[0]         # 第1ピクセルの色を取得
```

APA106 ドライバーは NeoPixel を継承していますが、内部的には異なる色順を使っています:

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

警告

デフォルトで `NeoPixel` はより一般的な *800kHz* のユニットを制御するように設定されています。NeoPixel オブジェクトを作成する際に `timing=0` を渡すことで、他のデバイス(よくあるのは 400kHz)を制御するために代替のタイミングを使えます。

NeoPixel の低レベル駆動については *machine.bitstream* を参照してください。この低レベルドライバは、デフォルトで RMT チャンネルを使います。この設定については *RMT.bitstream_channel* を参照してください。

APA102 (DotStar)はクロック端子が追加されているため、別のドライバーを使います。

静電容量タッチ

`machine` モジュールの `TouchPad` クラスを使います:

```
from machine import TouchPad, Pin

t = TouchPad(Pin(14))
t.read()              # タッチすると小さい数値を返す
```

`TouchPad.read` は容量変化に関連した値を返します。ピンにタッチすると小さい数字(通常は数十の値)となり、タッチしていない場合は大きい数字(1000を超える)となるのが一般的です。ただし、これらの値は相対的なもので、ボードと周囲の構成によって変わる可能性があるため、ある程度の調整が必要となるでしょう。

ESP32 で使える静電容量タッチ対応ピンは 0, 2, 4, 12, 13 14, 15, 27, 32, 33 の10本です。この他のピンに割り当てようとすると `ValueError` になります。

TouchPad を使ってESP32をスリープから復帰させることもできます:

```
import machine
from machine import TouchPad, Pin
import esp32

t = TouchPad(Pin(14))
t.config(500)                # ピンが接触したと見なす敷居値を設定
esp32.wake_on_touch(True)
machine.lightsleep()         # タッチされる MCU をスリープさせる
```

タッチパッドの詳細については <https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/peripherals/touch_pad.html>`_ を参照してください。

DHT ドライバー

DHT ドライバーはソフトウェアで実装され、すべてのピンで動作します:

```
import dht
import machine

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

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

WebREPL (Web ベースの対話プロンプト)

WebREPL (Web ブラウザ経由でアクセス可能な REPL)は、ESP32 ポートで使用可能な実験的な機能です。Web クライアントを https://github.com/micropython/webrepl (http://micropython.org/webrepl で入手可能なホストバージョン)からダウンロードしてきて、次のコマンドを実行して設定してください:

```
import webrepl_setup
```

画面の指示にしたがいます。再起動後、接続可能になります。起動時に自動起動を無効にした場合は、次のコマンドを使用してデーモンを実行することができます:

```
import webrepl
webrepl.start()

# もしくは指定のパスワードでスタート
webrepl.start(password='mypass')
```

WebREPL デーモンは STA または AP のいずれのアクティブインターフェースでも listen します。これにより、ルータ(STA インターフェイス)を介してでも、またはアクセスポイントに接続しているときでも直接 ESP32 に接続できます。

ターミナル/コマンドプロンプトでできることの他に、WebREPL にはファイル転送(アップロードとダウンロードの両方)の機能も用意しています。Webクライアントには、対応する機能のボタンがあります。また、上記のリポジトリのコマンドラインクライアント``webrepl_cli.py`` を使うこともできます。

ファイルを ESP32 ボードに転送するための、コミュニティでサポートされている他の代替方法については、MicroPython フォーラムを参照してください。