UNIT-II
MicroPython with ESP32

Introduction to ESP32

FEATURES OF ESP32:

- **Processor:**
  - CPU: Xtensa dual-core 32-bit LX6 microprocessor,600 DMIPS
  - Ultra low power (ULP) co-processor
  - Frequency: 80 MHz to 240 MHz
- **Memory:**
  - FLASH: 4MB
  - SRAM: 520 KB
- **Wireless connectivity:**
  - Wi-Fi: 802.11 b/g/n
  - Bluetooth: v4.2 and BLE (Bluetooth Low Energy)
- **Peripheral interfaces:**
  - 12-bit ADC up to 18 channels
  - 2 × 8-bit DACs
  - 10 × capacitive touch sensors
  - 4 × SPI
  - 2 × I2S interfaces
  - 2 × I2C interfaces
  - 3 × UART
  - 1 × Ethernet MAC interface
  - 1 × CAN bus 2.0
  - PWM up to 16 channels
  - Hall effect sensor
- **Security:**
  - IEEE 802.11 standard security features all supported, including WFA, WPA/WPA2 and WAPI
  - Secure boot
  - Flash encryption
- Wake up from GPIO interrupt, timer, ADC measurements, capacitive touch sensor interrupt
- Ultra-low-power management.

Digital input GPIO pins

It has six GPIO pins which can be used as digital input pins only. They cannot be configured as digital output pins. Hence, they do not have internally connected push-pull resistors. They can only be used as digital input pins.

- GPIO34
- GPIO35
- GPIO36
- GPIO37
- GPIO38
- GPIO39

All GPIO pins on initial boot-up or reset remains in active low state except the following pins. The following pins will be in active-high state by default during boot-up or reset. Therefore, you should initialize these pins to active-low state in the code in the setup function of the code.

- GPIO1
- GPIO3
- GPIO5
- GPIO6 to GPIO11
- GPIO14
- GPIO15

Analog to digital converter or Analog GPIO pins

This development board supports 18 ADC channels. And each channel is of 12 bits. So it has a good resolution. It can be used to measure analog voltage, current and any analog sensor which provides output in the form of analog voltage. These ADCs can also be used in sleep mode for lower power consumption. Each ADC channel has a resolution of 12 bits which is equal to 3.3 / 4095

Mapping of Analog pins with GPIO pins is shown below:

- ADC1_CH0 – GPIO36
- ADC1_CH1 – GPIO37
- ADC1_CH2 – GPIO38
- ADC1_CH3 – GPIO39
- ADC1_CH4 – GPIO32
- ADC1_CH5- GPIO33
- ADC1_CH6 – GPIO34
- ADC1_CH7 – GPIO35

- ADC2_CH0 – GPIO4
- ADC2_CH1 – GPIO0
- ADC2_CH2 – GPIO2
- ADC2_CH3 – GPIO15
- ADC2_CH4 – GPIO13
- ADC2_CH5 – GPIO12
- ADC2_CH6 – GPIO14
- ADC2_CH7 – GPIO27
- ADC2_CH8 – GPIO25
- ADC2_CH9 – GPIO26

Digital to Analog converter pins

This development board has two onboard integrated 8-bit DAC. DACs are used to convert digital signals into analog signals. DACs have many applications like voltage control and PWM control.

DAC_1 – GPIO25

DAC_2 – GPIO26

ESP32 UART Pins

According to the datasheet, ESP32 provides three universal asynchronous receivers and transmitter (UART) ports such as U0, U1, and U2. By default, only UART0 and UART2 can be used. To use UART1, we have to redefine the pins. Because default pins of UART1 such as GPIO9 and GPIO10 are internally connected to the SPI flash memory. Moreover, these pins are also not available on the ESP32 boards. Hence, we will have to reassign the pins for UART1 for serial communication. Luckily, the ESP32 board is capable to use almost all GPIO pins for serial connections. Here we have reassigned GPIO4 as RX pin and GPIO2 as TX pin.

UART Port	Rx	Tx
UART0	GPIO3	GPIO1
UART1	GPIO9	GPIO10
UART2	GPIO16	GPIO17

Touch sensor pins

ESP-WROOM-32 provide on board 10 capacitive touch sensors. So you don’t need to use separate touch sensors in your project when you are using this development board. These capacitive touch sensors can be used to detect any electrical and magnetic waves around like magnetic field detection. You can use a small array of pads instead of push buttons with these touch sensors.

- TOUCH0 – GPIO4
- TOUCH1 – GPIO0
- TOUCH2 – GPIO2
- TOUCH3 – GPIO15
- TOUCH4 – GPIO13
- TOUCH5 – GPIO12
- TOUCH6 – GPIO14
- TOUCH7 – GPIO27
- TOUCH8 – GPIO33
- TOUCH9 – GPIO32

Memory card interfacing pins

It also supports memory card interfacing through these pins.

- HS2_CLK – MTMS – GPIO14
- HS2_CMD – MTDO – GPIO15
- HS2_DATA0 – GPIO2
- HS2_DATA1 – GPIO4
- HS2_DATA2 – MTDI – GPIO12
- HS2_DATA3 – MTCK – GPIO13

External interrupt pins

All general purpose input output pins can be used as external interrupt. External interrupts are very useful. When you want to monitor change across any pin, you can use this pin as an interrupt instead of repeatedly monitoring the state of this pin.

PWM GPIO pins

All general purpose input output pins can be used to generate PWM except digital input pins from GPIO pins 34-39. Because these pins cannot be used as digital output pins. PWM signals are digital output signals. The maximum frequency of these PWM pins is 80MHz.

I2C communication pins

It has dedicated pins available for two-wire I2C communication. One pin is used for data transfer and another pin is used for clock synchronization.

- **GPIO21** is SDA pin.

- GPIO22 is SCL pin.

SPI Pins

By default, ESP32 has two SPI communication channels VSPI and HSPI and the following table provides the default SPI pins for both channels. But if we can also map these pins to other GPIO pins.

SPI Channel	MOSI	MISO	SCK/CLK	CS/SS
VSPI	GPIO23	GPIO19	GPIO18	GPIO15
HSPI	GPIO13	GPIO12	GPIO14	GPIO15

RTC pins

This board also provide RTC pins which can be used to trigger ESP32 from sleep mode.

- RTC_GPIO0 – GPIO36
- RTC_GPIO3 -GPIO39
- RTC_GPIO4 – GPIO34
- RTC_GPIO5 – GPIO35
- RTC_GPIO6 – GPIO25
- RTC_GPIO7 -GPIO26
- RTC_GPIO8 – GPIO33
- RTC_GPIO9 – GPIO32
- RTC_GPIO10 -GPIO4
- RTC_GPIO11 – GPIO0
- RTC_GPIO12 – GPIO2
- RTC_GPIO13 – GPIO15
- RTC_GPIO14 – GPIO13
- RTC_GPIO15 – GPIO12
- RTC_GPIO16 – GPIO14
- RTC_GPIO17 – GPIO27

Hall sensor pin

A complete guide on How to use built-in Hall Effect sensor of ESP32

It also has one hall sensor which is used to detect the magnetic field. Whenever you please this development board in the magnetic field, ESP32 generates a small voltage which can be measured with any pin.

I2S (Inter-IC Sound)

ESP32 contains two I2S serial communication peripherals. I2S is used for Audio transmission and reception. Each I2S controller works in a half-duplex communication mode. But we can combine these two available controllers to achieve full-duplex communication.

I2S -> GPIO25 and GPIO26 (I2S can be directly connected with DAC output channels (GPIO25 and GPIO26) to get direct Audio analog output).

Pulse Counter Module (PCNT)

ESP32 has 8 pulse counter (PCNT) modules which are used to count the number of positive or negative edges of the signal given to the GPIO pins. Each pulse counter module consists of a 16-bits counter register which counts from 0 to 65536 on the positive or negative edge of an input signal. Moreover, it can also be configured to count-up and count-down mode based on the edges of an input signal. Most importantly, it provides a control pin to enable or disable the count from an input signal.

All GPIO pins can be configured as an input signal for the pulse counter.

Remote Control Module

The remote control module driver can be used to transmit and receive IR remote control signals and any GPIO pin can be configured to receive and transmit IR signals.

Introduction to MicroPython

MicroPython is a Python 3 programming language re-implementation targeted for microcontrollers and embedded systems. MicroPython is very similar to regular Python. Apart from a few exceptions, the language features of Python are also available in MicroPython. The most significant difference between Python and MicroPython is that MicroPython was designed to work under constrained conditions.

Because of that, MicroPython does not come with the entire pack of standard libraries. It only includes a small subset of the Python standard libraries, but it includes modules to easily control and interact with the GPIOs, use Wi-Fi, and other communication protocols.

MicroPython is packed full of advanced features such as an interactive prompt, arbitrary precision integers, closures, list comprehension, generators, exception handling and more. Yet it is compact enough to fit and run within just 256k of code space and 16k of RAM.

MicroPython aims to be as compatible with normal Python as possible to allow you to transfer code with ease from the desktop to a microcontroller or embedded system.

MicroPython is a full Python compiler and runtime that runs on the ***bare-metal.*** You get an interactive prompt (the REPL) to execute commands immediately, along with the ability to run and import scripts from the built-in file system. The REPL has history, tab completion, auto-indent and paste mode for a great user experience.

Interactive REPL, or Read-Evaluate-Print Loop: This allows you to connect to a board and have it execute code without any need for compiling or uploading--perfect for quickly learning and experimenting with hardware!

Extensive software library: Like the normal Python programming language. MicroPython has libraries built in to support many tasks. For example, parsing JSON data from a web service, searching text with a regular expression, or even doing network socket programming is easy with built-in libraries for MicroPython.

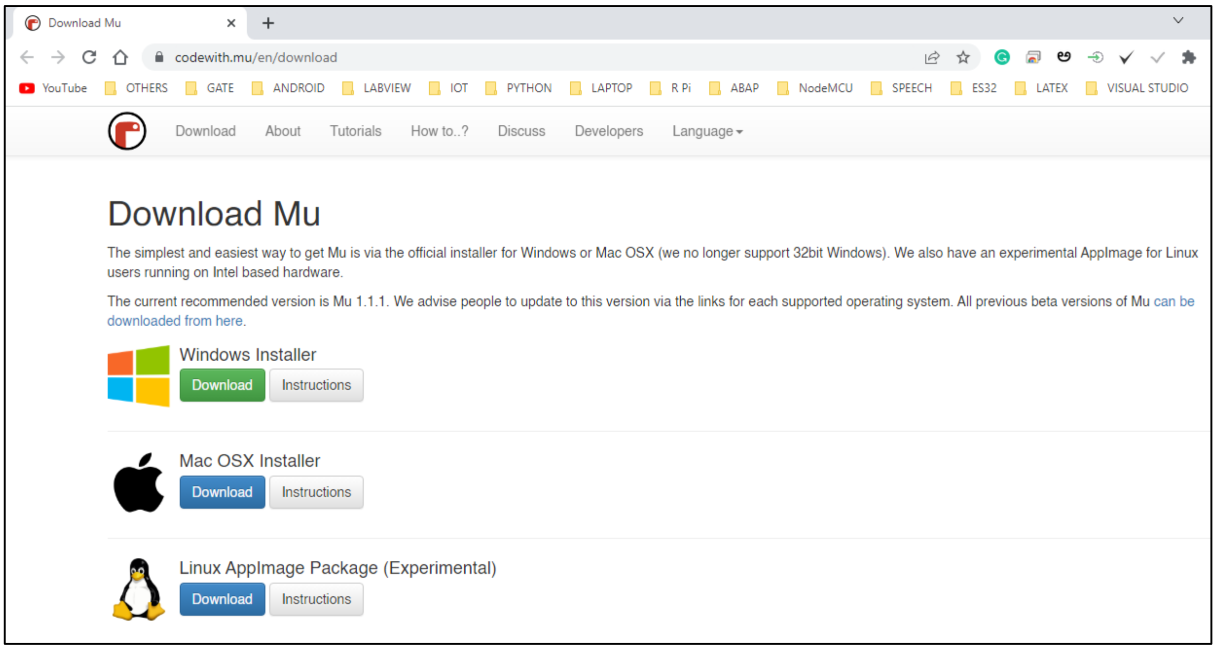
Extensibility: For advanced users MicroPython is extensible with low-level C/C++ functions so you can mix expressive high-level MicroPython code with faster low-level code when you need it.

Bare metal is a computer system without a base operating system (OS) or installed applications. It is a computer's hardware assembly, structure and components that is installed with either the firmware or basic input/output system (BIOS) software utility or no software at all.

Installing Mu Editor:

Installing Mu Editor is very straightforward. Follow the next steps:

1) Go to the following website and download Mu Editor for your operating system: https://codewith.mu/en/download.



Download Mu

The simplest and easiest way to get Mu is via the official installer for Windows or Mac OSX (we no longer support 32bit Windows). We also have an experimental AppImage for Linux users running on Intel based hardware.

The current recommended version is Mu 1.1.1. We advise people to update to this version via the links for each supported operating system. All previous beta versions of Mu can be downloaded from here.

Windows Installer
Download | Instructions

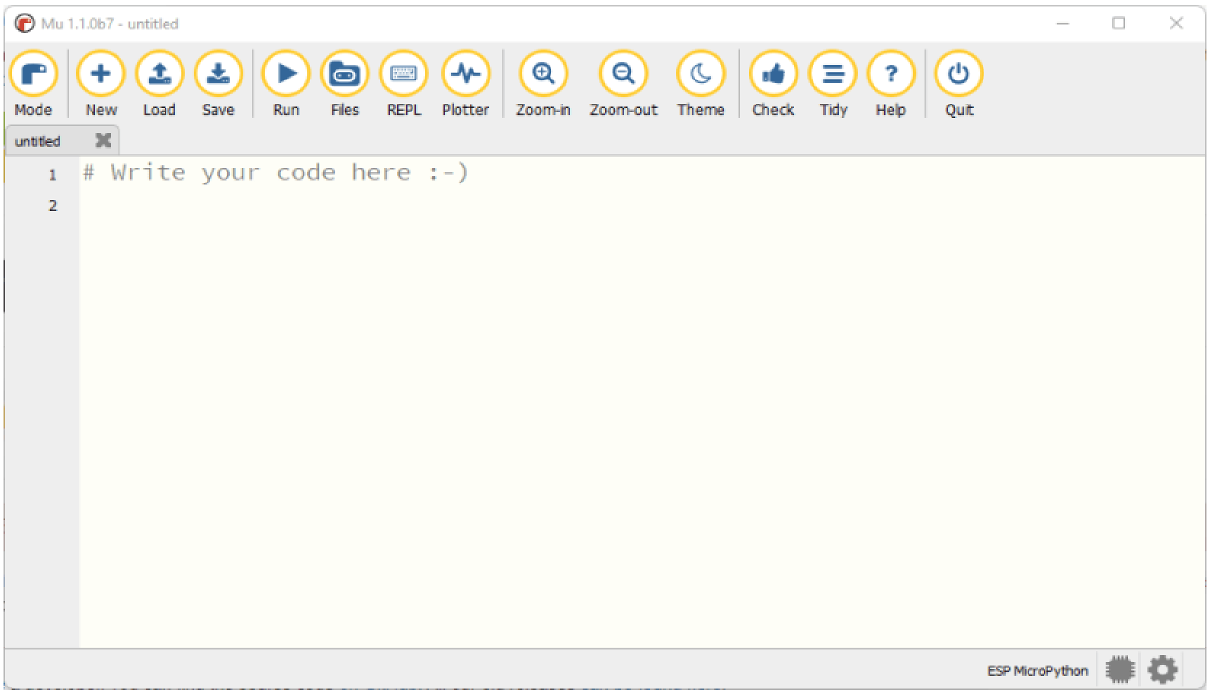
Mac OSX Installer
Download | Instructions

Linux AppImage Package (Experimental)
Download | Instructions

2) Run the installer you've just downloaded—it's probably in your *Downloads* folder.
3) Finally, follow the on-screen instructions to install it.

Mu Editor Overview

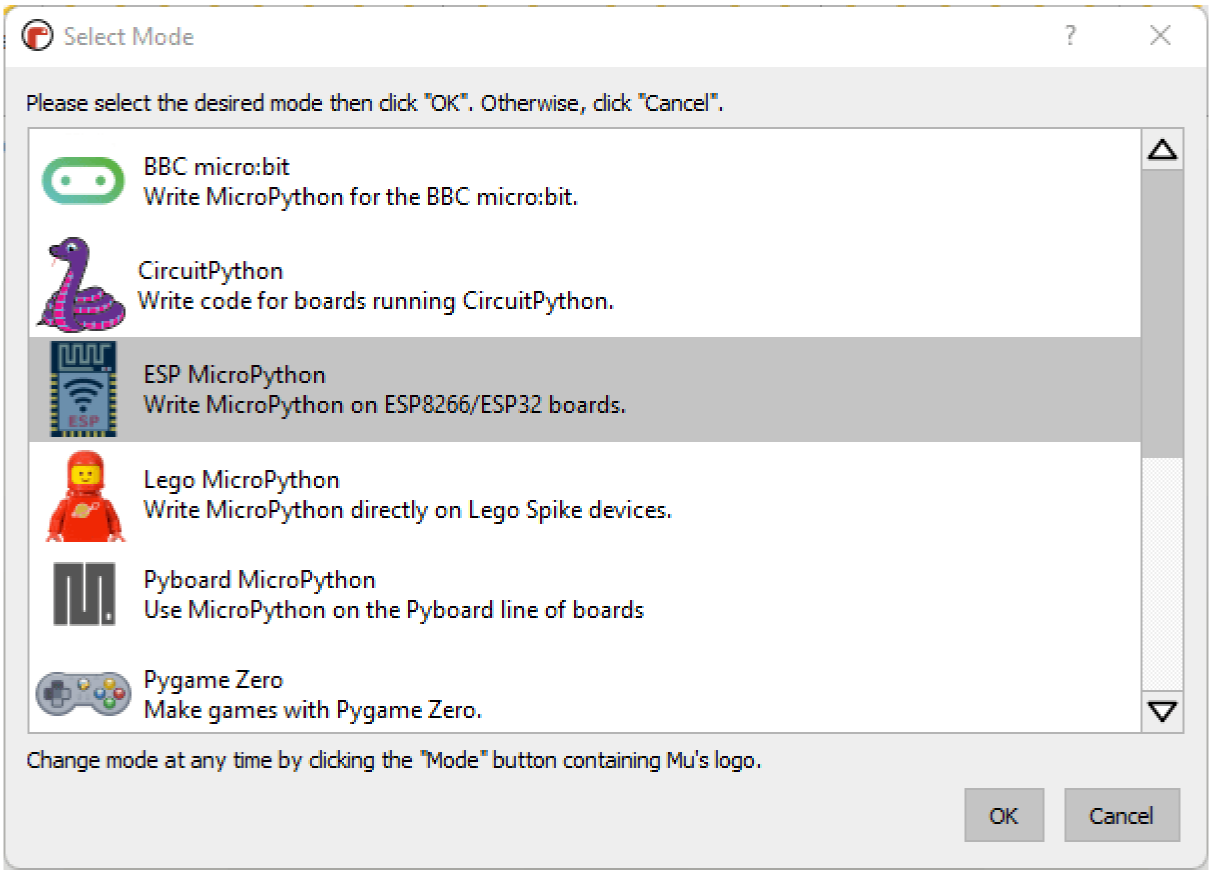
Open Mu Editor. The following window will open.



```
# Write your code here :-)
```

You can see that the interface is very simple and intuitive to use. Let's take a quick look at the buttons.

- **Mode**: choose which mode you want to use. Select ESP MicroPython



Select Mode

Please select the desired mode then click "OK". Otherwise, click "Cancel".

BBC micro:bit
Write MicroPython for the BBC micro:bit.

CircuitPython
Write code for boards running CircuitPython.

ESP MicroPython
Write MicroPython on ESP8266/ESP32 boards.

Lego MicroPython
Write MicroPython directly on Lego Spike devices.

Pyboard MicroPython
Use MicroPython on the Pyboard line of boards

Pygame Zero
Make games with Pygame Zero.

Change mode at any time by clicking the "Mode" button containing Mu's logo.

- **New**: creates a new file.
- **Load**: chooses a file to load into Mu.
- **Save**: saves the file to your computer.
- **Run**: runs the current script in your ESP32 or ESP8266 boards without actually uploading code to the board.
- **File**: opens a window with the files saved on your board filesystem and a window with the files saved on your computer's current directory.
- **REPL**: allows you to talk to the ESP32 or ESP8266 by writing commands that the board will run right away. REPL means **R**ead, **E**valuate, **P**rint, and **L**oop. The ESP32/ESP8266 waits for instructions by presenting >>> in the REPL. You should type your commands and hit Enter. The ESP will get the commands and send a response.
- **Plotter**: provides a graphical interface to display and plot values.
- **Zoom-in**: increases code font size.
- **Zoom-out**: decreases code font size.
- **Theme**: changes between different themes.

- **Check**: checks your code for errors.
- **Tidy**: tidies up your code when it comes to spaces and indentation.
- **Help**: opens Mu Editor Website help page.
- **Quit**: closes Mu Editor.

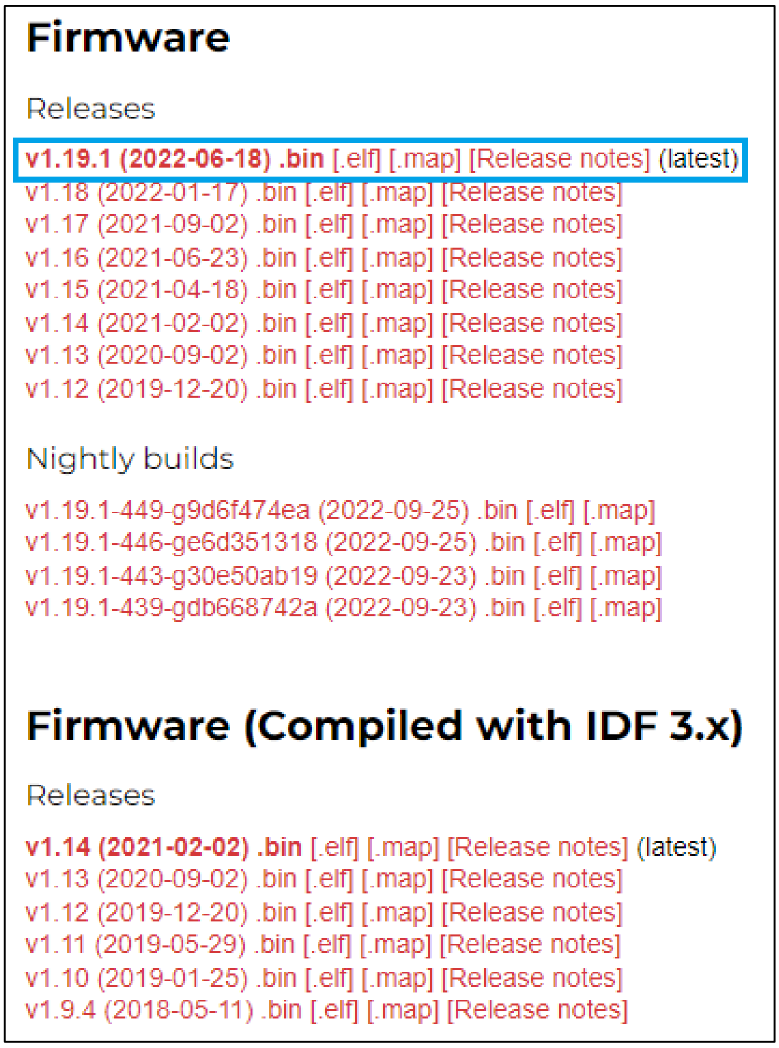
Flashing MicroPython Firmware using Mu-Editor

MicroPython isn't flashed onto the ESP32 or ESP8266 boards by default. The first thing is need to do to start programming your boards with MicroPython is flash/burn the firmware.

There are different ways in which you can do that. Mu Editor comes with a tool that allows you to quickly install MicroPython firmware on your board.

Downloading MicroPython ESP32 Firmware from below link
https://micropython.org/download/esp32/

⇨ You should see a similar web page (see figure below) with links to download .bin files. Download the latest release.



Firmware

Releases

v1.19.1 (2022-06-18) .bin [.elf] [.map] [Release notes] (latest)
v1.18 (2022-01-17) .bin [.elf] [.map] [Release notes]
v1.17 (2021-09-02) .bin [.elf] [.map] [Release notes]
v1.16 (2021-06-23) .bin [.elf] [.map] [Release notes]
v1.15 (2021-04-18) .bin [.elf] [.map] [Release notes]
v1.14 (2021-02-02) .bin [.elf] [.map] [Release notes]
v1.13 (2020-09-02) .bin [.elf] [.map] [Release notes]
v1.12 (2019-12-20) .bin [.elf] [.map] [Release notes]

Nightly builds

v1.19.1-449-g9d6f474ea (2022-09-25) .bin [.elf] [.map]
v1.19.1-446-ge6d351318 (2022-09-25) .bin [.elf] [.map]
v1.19.1-443-g30e50ab19 (2022-09-23) .bin [.elf] [.map]
v1.19.1-439-gdb668742a (2022-09-23) .bin [.elf] [.map]

Firmware (Compiled with IDF 3.x)

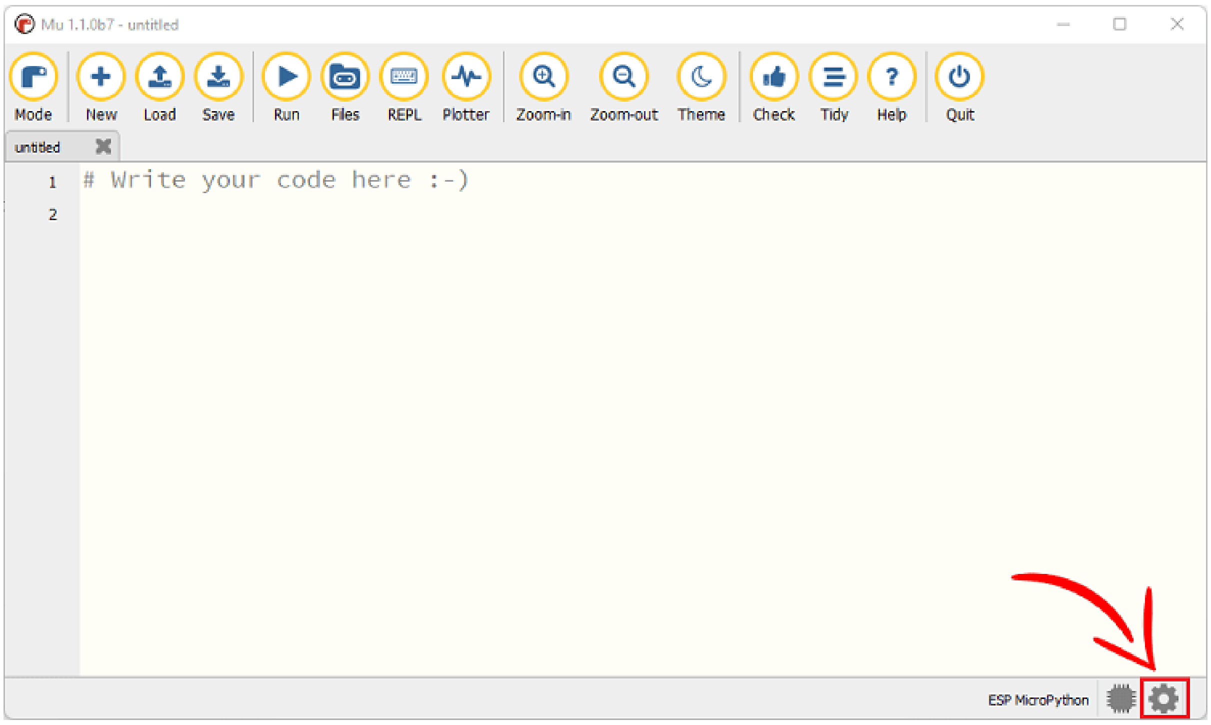
Releases

v1.14 (2021-02-02) .bin [.elf] [.map] [Release notes] (latest)
v1.13 (2020-09-02) .bin [.elf] [.map] [Release notes]
v1.12 (2019-12-20) .bin [.elf] [.map] [Release notes]
v1.11 (2019-05-29) .bin [.elf] [.map] [Release notes]
v1.10 (2019-01-25) .bin [.elf] [.map] [Release notes]
v1.9.4 (2018-05-11) .bin [.elf] [.map] [Release notes]

The downloaded file will probably go to the *Downloads* folder.

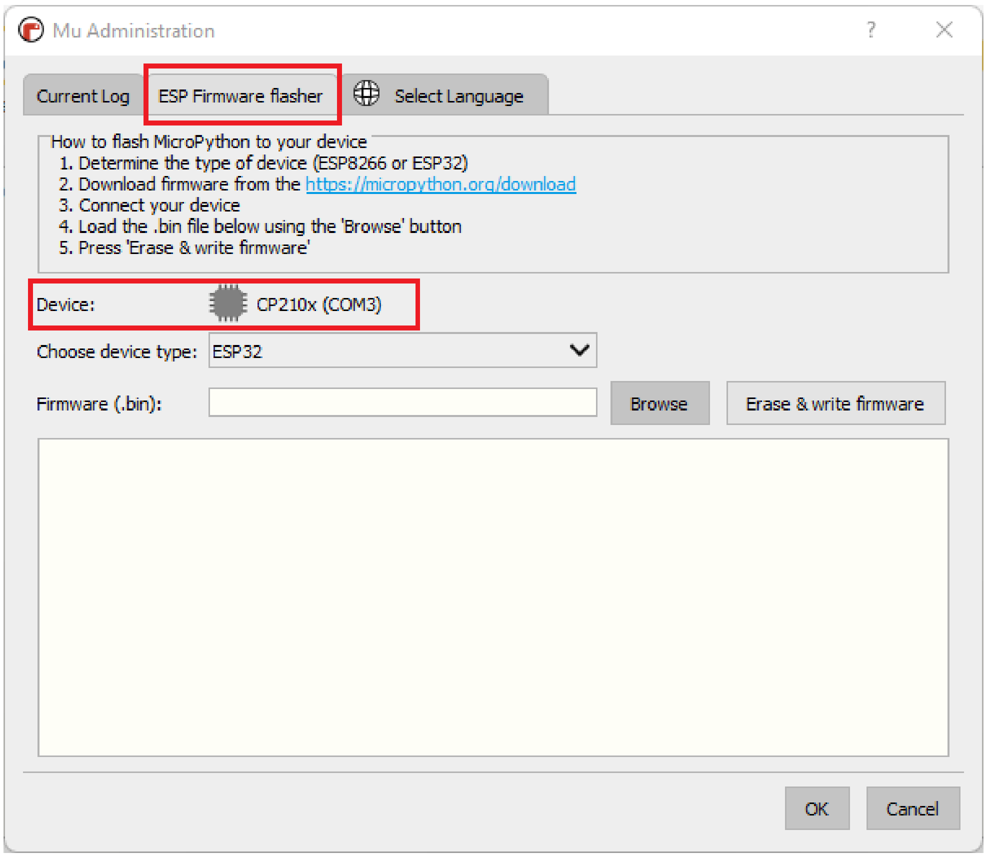
1) Connect ESP32 board to computer.

2) Open Mu Editor. And click on the gear(setting) icon at the bottom right corner.



```
# Write your code here :-)
```

3) Click on the **ESP Firmware flasher** tab. It should automatically detect an ESP32 board connected to a specific COM port.



How to flash MicroPython to your device
1. Determine the type of device (ESP8266 or ESP32)
2. Download firmware from the https://micropython.org/download
3. Connect your device
4. Load the .bin file below using the 'Browse' button
5. Press 'Erase & write firmware'

Device: CP210x (COM3)

Choose device type: ESP32

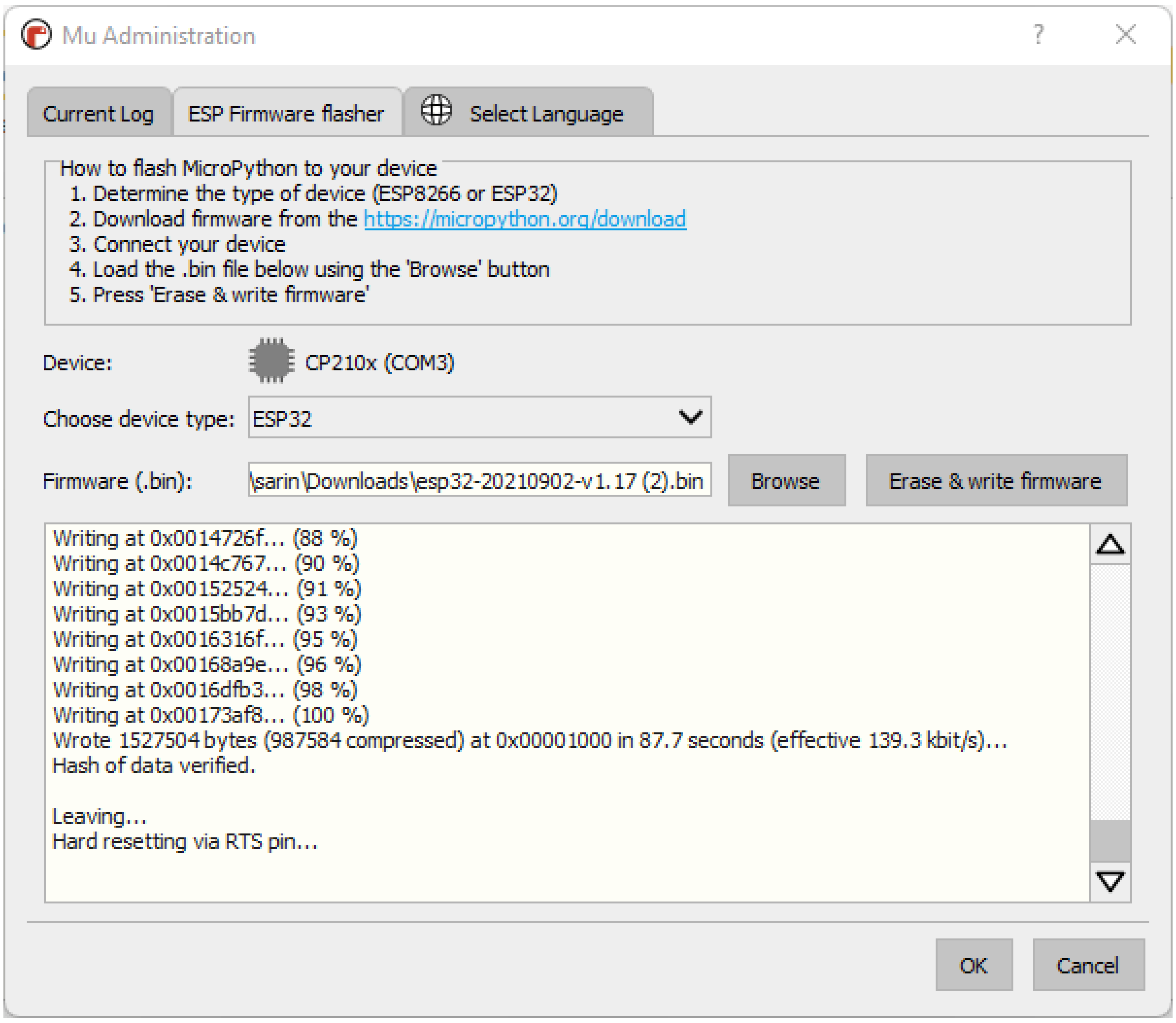
Firmware (.bin):

4) Choose device type: ESP32 board. Click on the **Browse** button and select the MicroPython firmware .bin file you've downloaded previously.

5) Click on **Erase & write firmware** to start burning MicroPython firmware.

6) For ESP32 board, it is need to hold the on-board BOOT button for about two seconds right after clicking on the Erase & Write firmware button to put board in flashing mode.

7) After a few seconds, the process is completed—see figure below.



How to flash MicroPython to your device
1. Determine the type of device (ESP8266 or ESP32)
2. Download firmware from the https://micropython.org/download
3. Connect your device
4. Load the .bin file below using the 'Browse' button
5. Press 'Erase & write firmware'

Device: CP210x (COM3)

Choose device type: ESP32

Firmware (.bin): \sarin\Downloads\esp32-20210902-v1.17 (2).bin

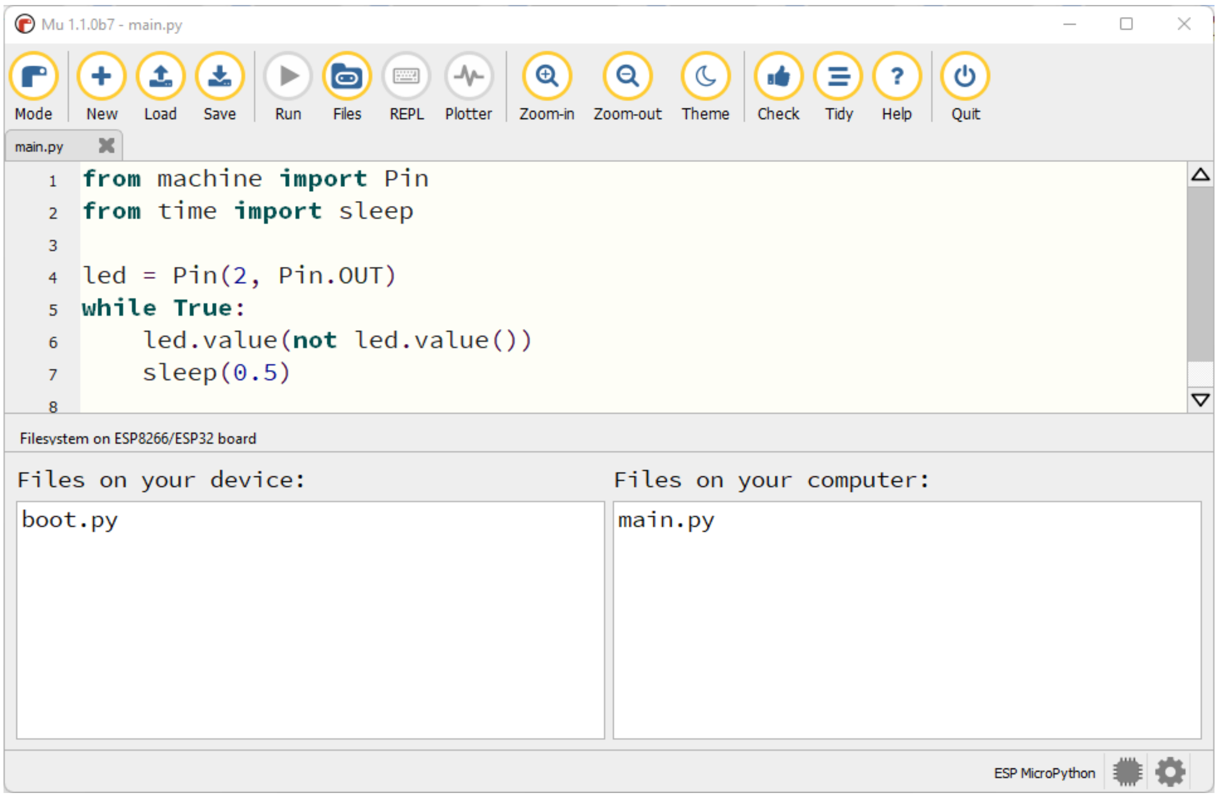
```
Writing at 0x0014726f... (88 %)
Writing at 0x0014c767... (90 %)
Writing at 0x00152524... (91 %)
Writing at 0x0015bb7d... (93 %)
Writing at 0x0016316f... (95 %)
Writing at 0x00168a9e... (96 %)
Writing at 0x0016dfb3... (98 %)
Writing at 0x00173af8... (100 %)
Wrote 1527504 bytes (987584 compressed) at 0x00001000 in 87.7 seconds (effective 139.3 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
```

Congratulations. You successfully flashed MicroPython firmware on your boards using Mu Editor.

Steps to run a code using Mu-Editor

1) Connect the ESP32 board to computer through a Micro-USB cable.
2) Open the Mu-Editor software.
3) Press the **New** button to create a new file.
4) Write the code in a newly created file.
5) Press the **Save** button to save the file to computer. Its recommend creating a project folder to save that file, for example, called *Blink*. Then, save the file inside that folder. **The file should be called main.py**.
6) Click on the **Files** button to open board file system and project directory on computer—see figure below. There should be a file called boot.py already on board. It is created by default when burn MicroPython firmware. There's also a file called main.py on computer with the code written in 4th step.



```
from machine import Pin
from time import sleep

led = Pin(2, Pin.OUT)
while True:
    led.value(not led.value())
    sleep(0.5)
```

Filesystem on ESP8266/ESP32 board

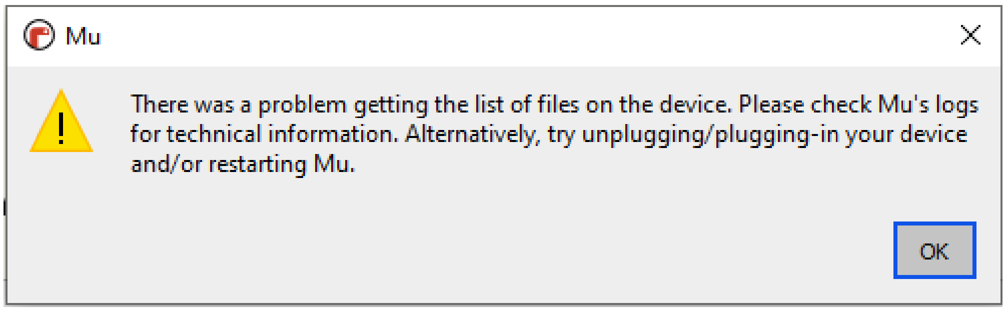
Files on your device:

boot.py

Files on your computer:

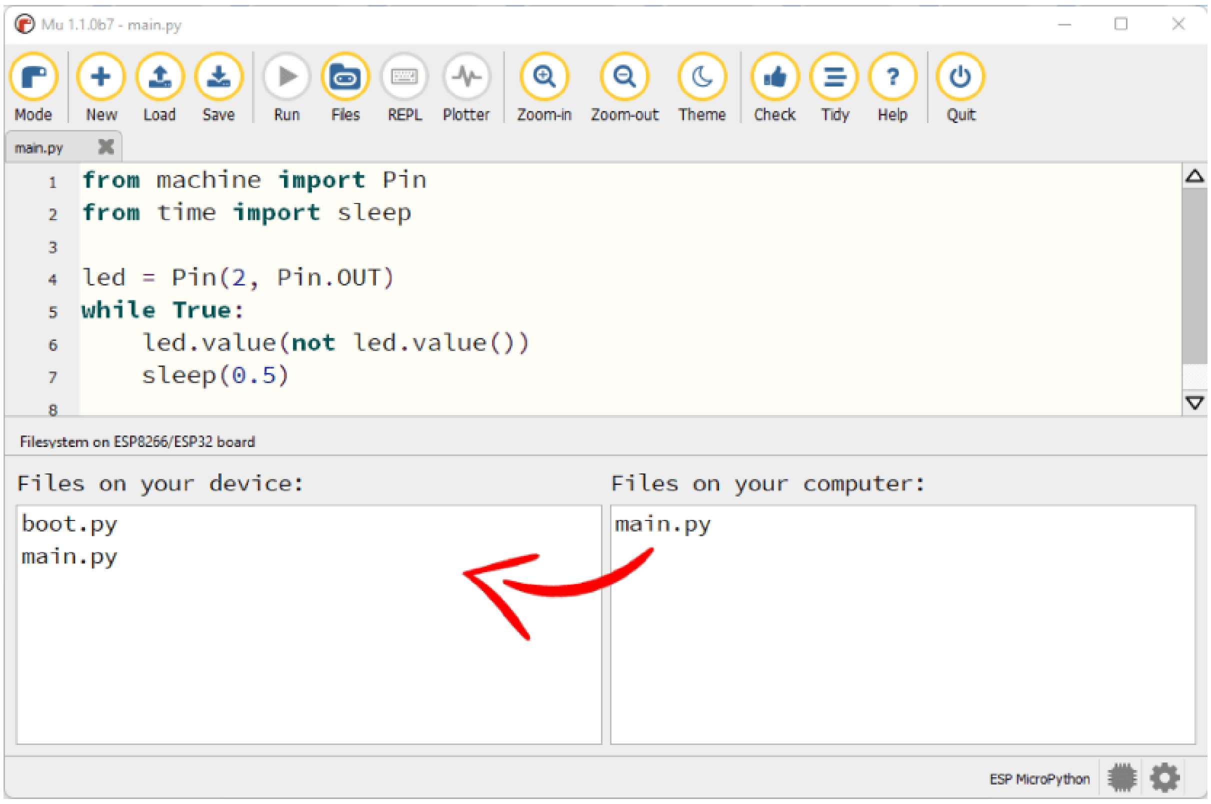
main.py

Troubleshooting: If you got an error when clicking on the **Files** button (see figure below), then click on the Files button again. Try this several times until you got a response.



There was a problem getting the list of files on the device. Please check Mu's logs for technical information. Alternatively, try unplugging/plugging-in your device and/or restarting Mu.

7) To upload code to your board, drag the main.py file from the “Files on your computer” window to the “Files on your device” window. Now, the main.py file is on your board filesystem.



```
from machine import Pin
from time import sleep

led = Pin(2, Pin.OUT)
while True:
    led.value(not led.value())
    sleep(0.5)
```

Note: Later, if you want to make changes to your main.py, you should save it on computer first. Then, drag the main.py again to the device window to overwrite the previous file.

8) Now, press the on-board ESP32 EN (ENABLE) button to restart board. After that, it will run the code we’ve just uploaded. The on-board blue LED should be blinking every half a second.

Program commands

Machine — functions related to the hardware

⇨ The machine module contains specific functions related to the hardware on a particular board.

⇨ Most functions in this module allow to achieve direct and unrestricted access to and control of hardware blocks on a system (like CPU, timers, buses, etc.).

⇨ Used incorrectly, this can lead to malfunction, lockups, crashes of your board, and in extreme cases, hardware damage.

Classes

- **class Pin – control I/O pins**
- class Signal – control and sense external I/O devices
- **class ADC – analog to digital conversion**
- class ADCBlock – control ADC peripherals
- **class PWM – pulse width modulation**
- **class UART – duplex serial communication bus**
- class SPI – a Serial Peripheral Interface bus protocol (controller side)
- class I2C – a two-wire serial protocol
- class I2S – Inter-IC Sound bus protocol
- class RTC – real time clock
- class Timer – control hardware timers
- class WDT – watchdog timer
- class SD – secure digital memory card (cc3200 port only)
- class SDCard – secure digital memory card

Delay and Timing

Use the **time** module

➢ **import time**

⇨ time.sleep(1)	# sleep for 1 second
⇨ time.sleep_ms(500)	# sleep for 500 milliseconds
⇨ time.sleep_us(10)	# sleep for 10 microseconds
⇨ start = time.ticks_ms()	# get millisecond counter
⇨ delta = time.ticks_diff(time.ticks_ms(), start)	# compute time difference

Pins and GPIO

Class Pin – control I/O pins

- A pin object is used to control I/O pins (also known as GPIO - general-purpose input/output).
- Pin objects are commonly associated with a physical pin that can drive an output voltage and read input voltages.
- The pin class has methods to set the mode of the pin (IN, OUT, etc.) and methods to get and set the digital logic level.

- **from machine import Pin**

```
p0 = Pin(0, Pin.OUT)                  # set GPIO0 as output pin
p0.on()                               # set pin to "on" (high) level
p0.off()                              # set pin to "off" (low) level
p0.value(1)                           # set pin to "on" (high) level
p2 = Pin(2, Pin.IN)                   # set GPIO2 as input pin
print(p2.value())                     # read and print the pin value
p4 = Pin(4, Pin.IN, Pin.PULL_UP)      # set GPIO4 as input pin with pull-up resistor
p5 = Pin(5, Pin.OUT, value=1)         # set GPIO5 as output pin with value "1"
p6 = Pin(6, Pin.OUT, drive=Pin.DRIVE_3)  # set maximum drive strength
```

- **Four drive strengths** are supported, using the **drive** keyword argument to the **Pin()** constructor, with different corresponding safe **maximum source/sink currents** and approximate **internal driver resistances:**

  - Pin.DRIVE_0: 5mA / 130 ohm
  - Pin.DRIVE_1: 10mA / 60 ohm
  - Pin.DRIVE_2: 20mA / 30 ohm (default strength if not configured)
  - Pin.DRIVE_3: 40mA / 15 ohm

- **Pin.value(x)**
  - This method allows to set and get the value of the pin, depending on the mode of the pin:
  - **Pin.IN -** The method returns the actual input value currently present on the pin.
  - **Pin.OUT -** The output buffer is set to the given value immediately.

- **Pin.on()**

  Set pin to “1” output level.

- **Pin.off()**

  Set pin to “0” output level.

UART

Class UART – duplex serial communication bus

The ESP32 has three hardware UARTs: UART0, UART1 **&** UART2

- **Class machine.UART(id, ...)**
  - Construct a UART object of the given id.

	UART0	UART1	UART2
tx	1	10	17
rx	3	9	16

UART.init(baudrate=9600,bits=8, parity=None, stop=1, *, ...)
Initialize the UART bus with the given parameters:
baudrate is the clock rate / is defines the data transfer rate in bits per second.
bits is the number of bits per character, 7, 8 or 9.
parity is the parity, None, 0 (even) or 1 (odd).
stop is the number of stop bits, 1 or 2.
Additional keyword-only parameters that may be supported by a port are:

- ⇨ **tx** specifies the TX pin to use.
- ⇨ **rx** specifies the RX pin to use.
- ⇨ **rts** specifies the RTS (output) pin to use for hardware receive flow control.
- ⇨ **cts** specifies the CTS (input) pin to use for hardware transmit flow control.
- ⇨ **txbuf** specifies the length in characters of the TX buffer.
- ⇨ **rxbuf** specifies the length in characters of the RX buffer.
- ⇨ **timeout** specifies the time to wait for the first character (in ms).
- ⇨ **timeout_char** specifies the time to wait between characters (in ms).

- **from machine import UART**
- uart1 = UART(1, 9600)   # init with given baudrate
- uart1 = UART(1, baudrate=9600, tx=33, rx=32) # init with given parameters
- uart1.init(baudrate=9600, bits=8, parity=None, stop=1)   # init with given parameters
- uart1.read(10)   # read upto 10 characters
- uart1.read()   # read all available characters
- uart1.readline()   # read a line
- uart1.readinto(buf)   # read and store into the given buffer
- uart1.write(‘hello')   # write the 5 characters

UART.deinit()

- Turn off the UART bus.

UART.any()

- Returns an integer counting the number of characters that can be available for read.
- It will return 0 if there are no characters available and a positive number if there are characters.
- The method may return 1 even if there is more than one character available for reading.

ADC

Class ADC – analog to digital conversion

The ADC class provides an interface to analog-to-digital convertors, and represents a single endpoint that can sample a continuous voltage and convert it to a discretised value.

- **from machine import ADC**
- adc = ADC(Pin(GPIO)) # create an ADC object on a GPIO
- val = adc.read() # read a raw analog value in the range 0-4095 for 12-bit resolution.
- val = adc.read_u16() # read a raw analog value in the range 0-65535
- val = adc.read_uv() # read an analog value in microvolts

The following line defines that we want to be able to read voltage in full range.

- **adc.atten(ADC.ATTN_11DB)**

This means its read voltage from 0 to 3.3V. This corresponds to setting the attenuation ratio of 11db. For that, we use the atten () method and pass as argument: ADC.ATTN_11DB.

- **The atten() method can take the following arguments:**
- ADC.ATTN_0DB — the full range voltage: 0 to 1.2V (0 to 4095 for 12bit resolution)
- ADC.ATTN_2_5DB — the full range voltage: 0 to 1.5V (0 to 4095 for 12bit resolution)
- ADC.ATTN_6DB — the full range voltage: 0 to 2.0V (0 to 4095 for 12bit resolution)
- ADC.ATTN_11DB — the full range voltage: 0 to 3.3V (0 to 4095 for 12bit resolution)

- When analog input value changes, we get values from 0 to 4095 – that's because the ADC pins have a 12-bit resolution by default.
- If we want to get values in other ranges the we have to the change the resolution using the **width()** method as follows:
  - **ADC.width(bit)**

The bit argument can be one of the following parameters:

- ADC.WIDTH_9BIT: range 0 to 511
- ADC.WIDTH_10BIT: range 0 to 1023
- ADC.WIDTH_11BIT: range 0 to 2047
- ADC.WIDTH_12BIT: range 0 to 4095

PWM

Class PWM – Pulse Width Modulation

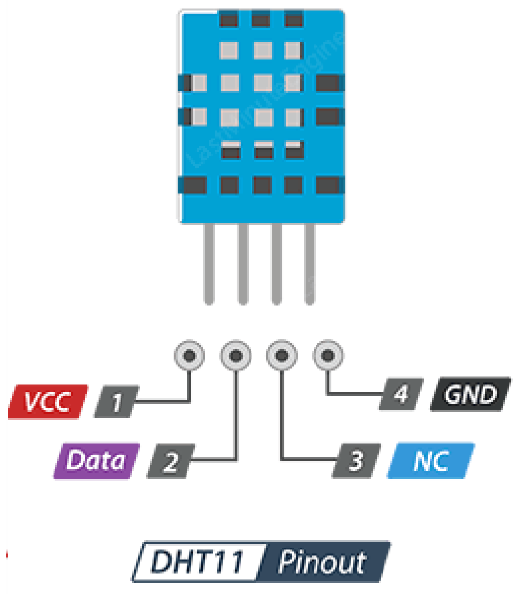
PWM can be enabled on all output-enabled pins. The base frequency can range from 1Hz to 40MHz but there is a tradeoff (compromise); as the base frequency increases the duty resolution decreases.

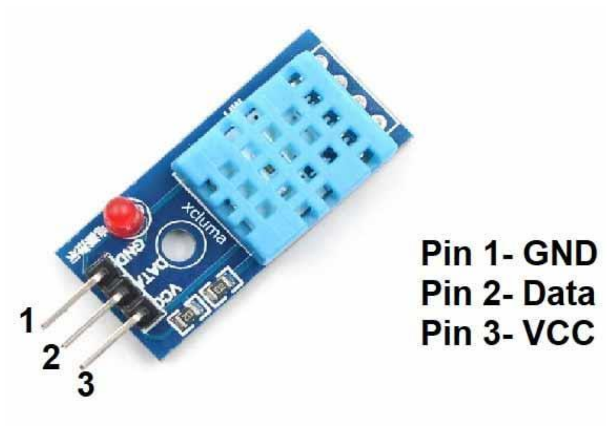
- **from machine import Pin, PWM**

```
⇨ pwm = PWM(Pin(GPIO))        # create PWM object on a pin GPIO
⇨ freq = pwm.freq()           # get current frequency (default 5kHz)
⇨ pwm.freq(1000)              # set PWM frequency from 1Hz to 40MHz
⇨ duty = pwm.duty()           # get current duty cycle, range 0-1023 (default 512, 50%)
⇨ pwm.duty(256)               # set duty cycle from 0 to 1023 as a ratio duty/1023, (now 25%)
⇨ duty_u16 = pwm.duty_u16()          # get current duty cycle, range 0-65535
⇨ pwm.duty_u16(2**16*3//4)           # set duty cycle from 0 to 65535 as a ratio
                                       duty_u16/65535, (now 75%)
⇨ duty_ns = pwm0.duty_ns()           # get current pulse width in ns
⇨ pwm0.duty_ns(250_000)              # set pulse width in nanoseconds from 0
  to 1_000_000_000/freq, (now 25%)
⇨ pwm0.deinit()                                  # turn off PWM on the pin
⇨ pwm2 = PWM(Pin(2), freq=20000, duty=512)       # create and configure in one go
⇨ print(pwm2)                                    # view PWM settings
⇨ pwm.init(freq=5000, duty_ns=5000)              #initialize
⇨ pwm.duty_ns(3000)                              # set pulse width to 3us
```

DHT11 Sensor

- The **DHT11** is a basic, low cost digital temperature and humidity sensor.
- **DHT11** is a single wire digital humidity and temperature sensor, which provides humidity and temperature values serially with one-wire protocol.
- **DHT11** sensor provides relative humidity value in percentage (20 to 90% RH-Relative Humidity) and temperature values in degree Celsius (0 to 50 °C).
- **RH of Air**: Determines the amount of water vapor present in air.
- **VCC:** Power supply 3.3 to 5.5 Volt DC.
- **DATA:** Digital o/p pin is used to communication b/w the sensor and the microcontroller.
- **NC:** Not connected
- **GND:** should be connected to the ground of ESP32.





```
➢ import dht
➢ import machine
➢ d = dht.DHT11(machine.Pin(GPIO))# create DHT11 object on a pin GPIO
➢ d.measure()                      # measures the DHT11 data
➢ d.temperature()                  # eg. 23 (°C)
➢ d.humidity()                     # eg. 41 (% RH)
➢ d = dht.DHT22(machine.Pin(GPIO)) # create DHT22 object on a pin GPIO
➢ d.measure()                      # measures the DHT22 data
➢ d.temperature()                  # eg. 23.6 (°C)
➢ d.humidity()                     # eg. 41.3 (% RH)
```