ESP8266 & Micropython

By jean-claude.feltes@education.lu

1. Micropython installation and deployment

https://docs.micropython.org/en/latest/esp8266/tutorial/intro.html

Downloads page:
http://micropython.org/download#esp8266

Install the uPython firmware:
http://docs.micropython.org/en/latest/esp8266/tutorial/intro.html#deploying-the-firmware

```
(sudo) pip(3) install esptool
```

Using esptool.py you can erase the flash with the command:

```
esptool.py --port /dev/ttyUSB0 erase_flash
```

Then deploy the new firmware by opening a terminal in the folder containing the firmware using:

```
esptool.py --port /dev/ttyUSB0 --baud 460800 write_flash --flash_size=detect 0
esp8266-<xxxxx>.bin
```

(Replace <xxxxx> by the correct reference number of the downloaded firmware.)

2. Serial / USB communication with the ESP8266

A board with a USB connector (having a USB to serial converter) can be directly accessed by opening a serial terminal like GtkTerm with the settings
Port: /*dev*/ttyUSB0, 115200baud

After a hardware reset, the terminal displays something like:

```
MicroPython v1.11-8-g48dcbbe60 on 2019-05-29; ESP module with ESP8266
Type "help()" for more information.
>>>
```

3. Interactive Python (REPL) on the ESP8266

What the hack is REPL? It stands for: Read, Evaluate, Print and Loop. The process is:

1. take user input.
2. evaluate the input.
3. print the output to the user.
4. repeat.

So it is simply the language shell, interactive Python.

Example:

```
>>> for i in range(0,3):
...     print(i)
...
...
... http://micropython.org/webrepl/
0
1
2
>>>
```

There is one thing to know: you leave an indented block by pressing 3 times <Enter>, or with <Backspace> to leave the indentation level, and <Enter> for next line, until you are back to the Python prompt “>>>”

```
>>> def test():
...     for x in range(0,5):
...         print(x)
...
>>> test()
0
1
2
3
4
>>>
```

4. Web REPL

Interactive Python can also be done over the air, with a WiFi connection.

To set up the Web REPL, run this via serial terminal:

```
>>> import webrepl_setup
WebREPL daemon auto-start status: disabled

Would you like to (E)nable or (D)isable it running on boot?
(Empty line to quit)
> E
To enable WebREPL, you must set password for it
New password (4-9 chars): 8266
Confirm password: 8266
Changes will be activated after reboot
Would you like to reboot now? (y/n) y
>>>
 ets Jan  8 2013,rst cause:2, boot mode:(3,6)

load 0x40100000, len 31024, room 16
tail 0
chksum 0xb4
…
WebREPL daemon started on ws://192.168.4.1:8266
Started webrepl in normal mode

MicroPython v1.11-8-g48dcbbe60 on 2019-05-29; ESP module with ESP8266
Type "help()" for more information.
>>>
```

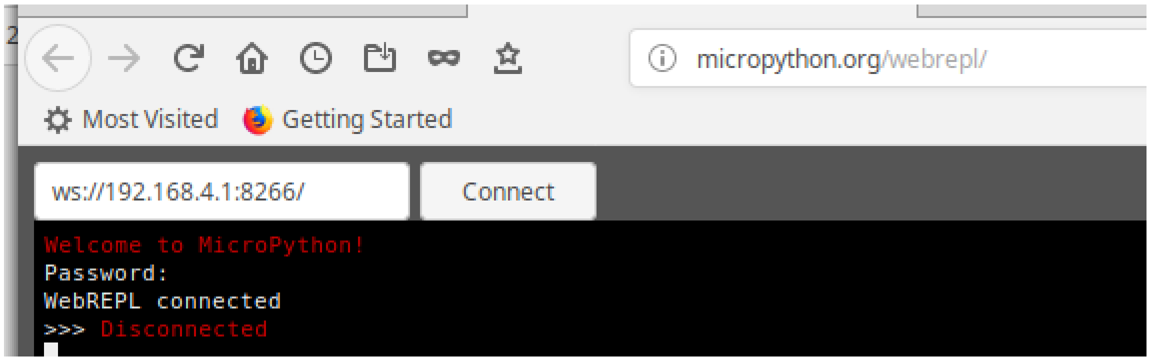
After this, the ESP8266 starts an access point named MicroPython-<number>.

You can connect to it with the **password “micropythoN”.** (Uppercase “N”!)

The password set for WebREPL is not the WiFi password, but the password for the Web REPL client. So you have two different passwords to use.

You can use the client at http://micropython.org/webrepl

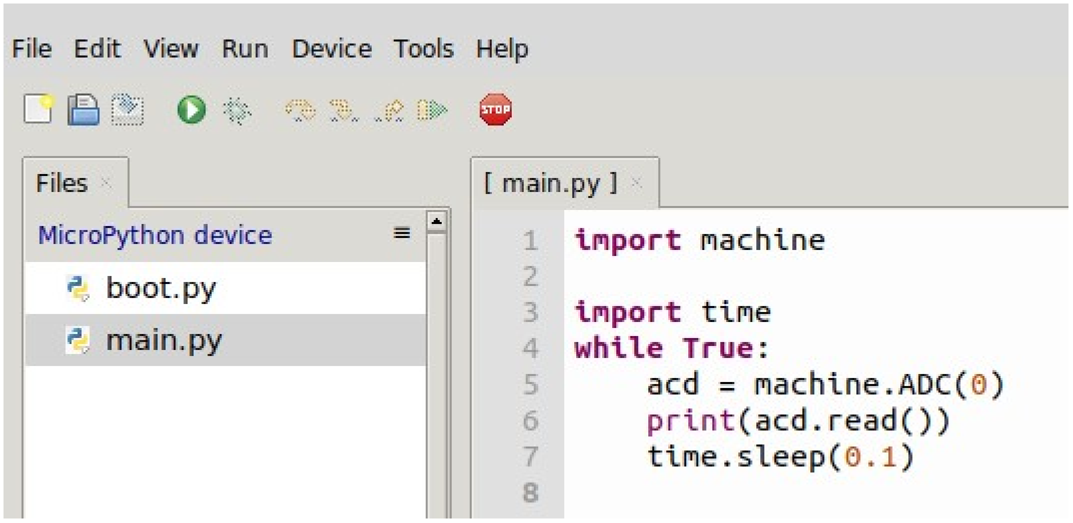
or download a local client at https://github.com/micropython/webrepl



```
Welcome to MicroPython!
Password:
WebREPL connected
>>> Disconnected
```

Now interactive Python can be done in the browser, just like through the serial connection.

5. Is there an IDE for efficient programming? Yes: Thonny for example



```
import machine

import time
while True:
    acd = machine.ADC(0)
    print(acd.read())
    time.sleep(0.1)
```

The new version of Thonny allows up- and downloading programs to and from the ESP8266.

While the interactive style is fine for trying short one liners, the IDE is really nice for writing longer programs.

- In the “Run” menu, go to “Select Interpreter” and select MicroPython

- The files on the device are shown (if there are any), and they can be downloaded and displayed in the main window.
- Editor content can directly be run on the chip with “Run current script” or <F5> or the green arrow button
- REPL is still possible in the Shell window

6. Files

Writing a file:

```
>>> f=open ("test.dat", "w")
>>> f.write("HELLO")
5
>>> f.write("3.14")
4
>>> f.close()
```

The numbers 5 and 4 are the numbers of bytes written.

This works for strings.

This does not work:

```
>>> x=3.14
>>> f.write(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object with buffer protocol required
```

Reading a file:

```
>>> f=open("test.dat", "r")
>>> f.read()
'HELLO3.14'
>>> f.close()
```

This example shows that the write method does not automatically insert line breaks.

If needed, this has to be done manually: `f.write("Hello\n")`

Directory listing:

```
>>> import os
>>> os.listdir()
['boot.py', 'webrepl_cfg.py', 'test.dat']
>>>
```

Create and remove directories:

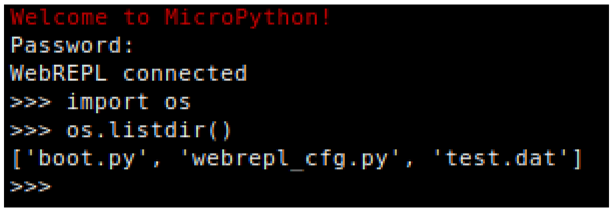
```
>>> os.mkdir('mydir')
>>> os.listdir()
['boot.py', 'webrepl_cfg.py', 'test.dat', 'mydir']
>>> os.rmdir('mydir')
>>> os.listdir()
['boot.py', 'webrepl_cfg.py', 'test.dat']
>>>
```

7. Start up scripts

There are two files that are treated specially by the ESP8266 when it starts up: boot.py and main.py. The boot.py script is executed first (if it exists) and then once it completes the main.py script is executed. You can create these files yourself and populate them with the code that you want to run when the device starts up. (See appendix for an example)

8. Getting and writing files through WebREPL

- Connect to the WiFi MicroPython-<number> with password "micropythoN"
- Open the WebREPL client http://micropython.org/webrepl/ in a browser (or connect with the local client)
- Connect with password set in webrepl_setup ("8266" in our example)
- To list available files, do on the prompt:



```
Welcome to MicroPython!
Password:
WebREPL connected
>>> import os
>>> os.listdir()
['boot.py', 'webrepl_cfg.py', 'test.dat']
>>>
```

- To get a file, use the "Get a file" button. You can choose to open the file with a text editor.
- To upload a file, use the "Send a file" button.

9. Using custom functions in custom modules

With the WebREPL it is easy to upload custom functions, to avoid too much typing in the interactive mode.

Example:

The file test.py contains the function myfunction()

```
def myfunction():
    for i in range(0,5):
        print('HELLO')
```

After uploading it, the module can be imported and the function executed:

```
>>> import test
>>> test.myfunction()
HELLO
HELLO
HELLO
HELLO
HELLO
```

The same way, anything can be executed at boot, when the appropriate statements are contained in the boot.py or the main.py file.

Note that when a changed version of a module should be imported, it must first be deleted from memory and then reimported, see Appendix.

10. Automatically starting a program at boot

Use WebRPL or Thonny to save the program **main.py** in the file system.

main.py is executed at reboot.

11. Using GPIOs

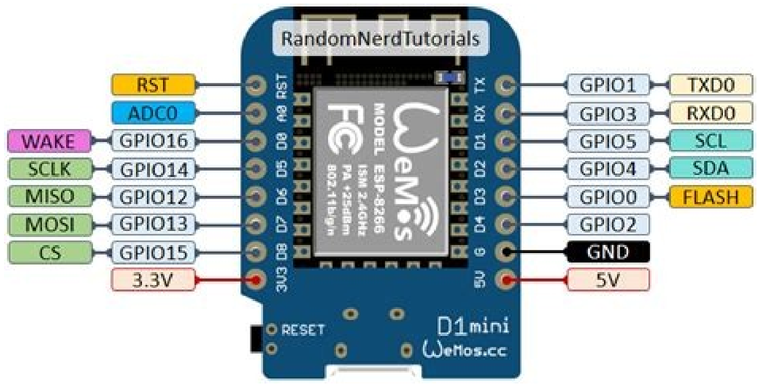


Image from randomnerdtutorials.com

Note that the pin numbers in Micropython are the chip pin numbers that do not correspond to the pin numbers like D0 etc. marked on the boards.

Take care: the logic level is 3.3V!

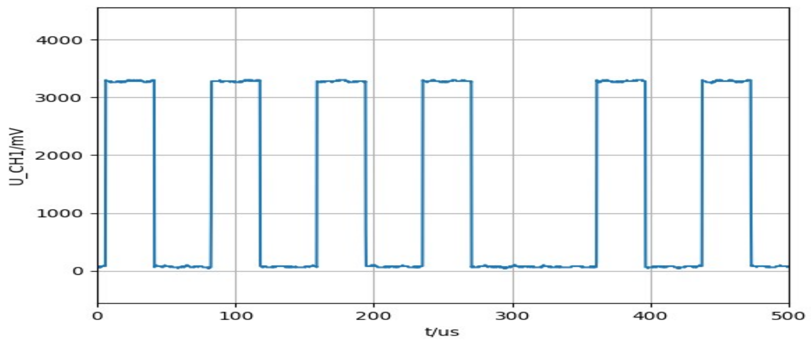
Available pins are: 0, 1, 2, 3, 4, 5, 12, 13, 14, 15, 16, which correspond to the actual GPIO pin numbers of ESP8266 chip.

```
from machine import Pin
>>> d4 = Pin(2, Pin.OUT)
>>> d4.on()
>>> d4.off()
```

Now how fast will this be done? Let's test with a loop and the oscilloscope:

```
 while True:
...     d4.on()
...     d4.off()
```

The pulses and pauses are about 40µs long, so not quite as slow as expected. Unfortunately there is some jitter:



Input pins can be configured alike, with the interesting possibility to use an internal pullup resistor:

```
>>> from machine import Pin
>>> pin = Pin(0, Pin.IN)
```

or

```
pin = Pin(0, Pin.IN, Pin.PULL_UP)
```

The pin state can be read like this:

```
>>> s = pin.value()
>>> print(s)
1
```

12. Time

The time module has the functions sleep() for slow delays (in seconds), and sleep_ms and sleep_us for delays in the miliisecond and microsecond range.

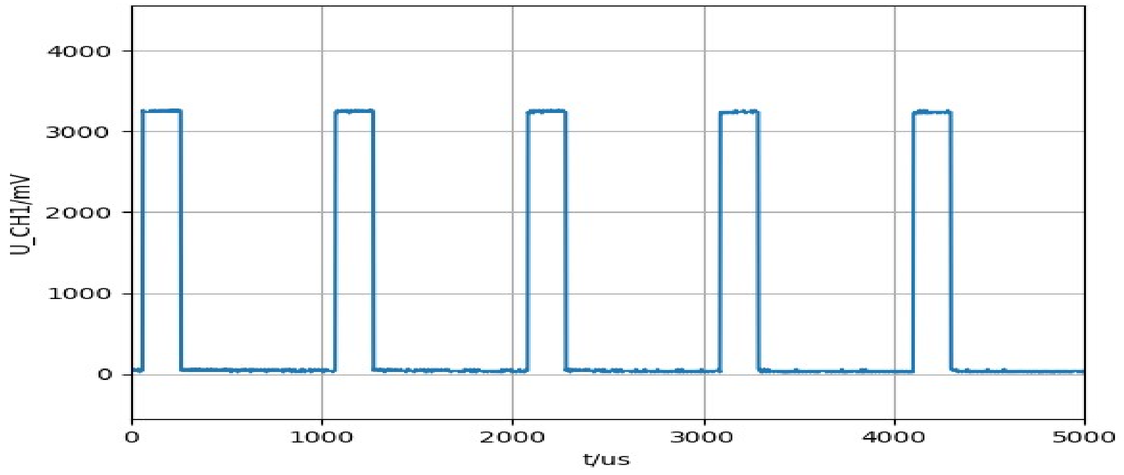
I tested these with loops:

```
from machine import Pin
from time import sleep, sleep_ms, sleep_us

def loop_1Hz():
    d4 = Pin(2, Pin.OUT)
    while True:
        d4.on()
        sleep(0.5)
        d4.off()
        sleep(0.5)

def loop_100Hz():
    d4 = Pin(2, Pin.OUT)
    while True:
        d4.on()
        sleep_ms(5)
        d4.off()
        sleep_ms(5)

def loop_1kHz():
    d4 = Pin(2, Pin.OUT)
    while True:
        d4.on()
        sleep_us(500)
        d4.off()
        sleep_us(500)

loop_1kHz()          # or loop_100Hz or loop_1Hz
```

As expected the slow loops give a good accuracy, but not so the 1kHz signal that has a lot of jitter.
This is seen on the oscilloscope, and heard acoustically if the signal is sent to a speaker (by the way, this is an excellent method to detect jitter that is not even seen on the oscilloscope).

13. PWM

```
>>> import machine
>>> p0 = machine.Pin(0)
>>> pwm0 = machine.PWM(p0)          # pin D3
>>> pwm0.freq(1000)
>>> pwm0.duty(200)                  # 0...1023  (10 bit)
```



On the ESP8266 the pins 0, 2, 4, 5, 12, 13, 14 and 15 all support PWM. The limitation is that they must all be at the same frequency, and the frequency must be between 1Hz and 1kHz.
Current frequency and duty cycle can be found like this:

```
>>> pwm0.freq()
1000
>>> pwm0.duty()
200
```

To stop PWM:

```
pwm0.deinit()
```

The PWM signal is nice with practically no jitter, as confirmed acoustically.

14. Controlling a servo

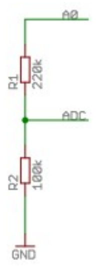
This can be done with a PWM signal by setting the frequency to 50Hz.

The duty range is from 40...115.

15. ADC: Reading analog values

The ESP8266 has only one analog input on pin A0.

The voltage range is 0...1V corresponding to values 0...1023 (10 bit).



The WEMOS D1 mini (and some others) has a voltage divider at the ADC input, so that the range is 320/100 * 1V = 3.2V, approximately corresponding to the 3.3V supply voltage.
Input is at the pin A0 of the board.

Read the ADC and print the voltage in Volts:

```
import machine
import time

while True:
    adc = machine.ADC(0)
    v = adc.read() / 1024 * 3.2
    print(v)
    time.sleep(0.1)
```

16. Timer interrupts

For the ESP8266 only virtual timers are supported, that are based on the underlying Real Time Operating System (RTOS). The ESP32 also supports hardware timers.

http://docs.micropython.org/en/latest/reference/isr_rules.html

https://techtutorialsx.com/2017/10/07/esp32-micropython-timer-interrupts/

Take care:
Timer interrupts may continue after termination of the Python program.
I experienced a chip that no longer responded to any command.

If you have this problem, reflash micropyton with esptool:

1. erase the flash
2. flash micropython

like this:

```
esptool.py --port /dev/ttyUSB0 erase_flash
esptool.py --port /dev/ttyUSB0 --baud 460800 write_flash --flash_size=detect 0
esp8266-<xxxxx>.bin
```

Example test.py:

```
from machine import Timer

tim = Timer(-1)
tim.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print("HELLO"))
```

This creates an interrupt that writes “HELLO” to the serial interface every 2000ms = 2s.
Timer(-1) means that a software timer is used.

Instead of `mode=Timer.PERIODIC`, `mode=Timer.ONE_SHOT` could be used for a single event.

This program continues to run even after the Thonny STOP button is pushed.
We can verify that a “Disconnect” in Thonny does not change this behaviour. If connected to a serial terminal, the “HELLO” messages continue to display every 2s.

Now how can we stop this? By pushing the reset button on the board.
But take care that your test program is not named “main.py”, otherwise it will be restarted at boot!

A better idea is to **stop the timer before leaving the program**. This is done with **Timer.deinit().**

The following example writes “Hello” every 2s by interrupt. The loop waits for a button push on pin D3 = GPIO0. This ends the program and the timer interrupt.

```
from machine import Timer
from machine import Pin
pin = Pin(0, Pin.IN, Pin.PULL_UP)

def printhello(e):
    print("Hello")

tim = Timer(-1)
tim.init(period=2000, mode=Timer.PERIODIC, callback=printhello)

while pin.value() != 0:
    pass

print ("Exit loop")
tim.deinit()
```

Here the interrupt service routine is a separate function printhello that has one dummy parameter e that is not used. (Every Python callback function has an event parameter that specifies the event, e.g. mouse click or keybaoard character in a GUI application)

The pin is initialised with an internal pullup rsistor, so a button to ground can be used. On button push we leave the loop and deinit the timer at program end.

Important note:
The interrupt service routine should be as short as possible. Often it is better to only set a flag that is evaluated in a loop in the main program.

More on this is found here:

https://docs.micropython.org/en/latest/reference/isr_rules.html

17. Appendix

Files on my test configuration after installing WebREPL

boot.py

```
# This file is executed on every boot (including wake-boot from deepsleep)
#import esp
#esp.osdebug(None)
import uos, machine
#uos.dupterm(None, 1) # disable REPL on UART(0)
import gc
import webrepl
webrepl.start()
gc.collect()
```

webrepl_cfg.py

```
PASS = '8266'
```

Writing custom libraries:

https://micropython-dev-docs.readthedocs.io/en/latest/adding-module.html

Quick reference for the ESP8266:

https://docs.micropython.org/en/latest/esp8266/quickref.html

About modules:

https://www.programiz.com/python-programming/modules

Remove modules from memory

If you are working on a library, once the library module is imported, it is not changed by a subsequent import statement. To change it, it must first be deleted and then imported again.

In the following example, the module ‘test’ has changed, and we want to reimport it:

```
>>> import sys
>>> sys.modules
{'webrepl': <module 'webrepl'>, 'test': <module 'test'>, 'flashbdev': <module
'flashbdev'>, 'websocket_helper': <module 'websocket_helper'>, 'webrepl_cfg': <module
'webrepl_cfg'>}

>>> del sys.modules['test']

>>> sys.modules
{'webrepl': <module 'webrepl'>, 'flashbdev': <module 'flashbdev'>,
'websocket_helper': <module 'websocket_helper'>, 'webrepl_cfg': <module
'webrepl_cfg'>}
```

The **sys.modules** command allows us to see which modules are present, and the module ‘test’ is deleted with **del sys.modules['test']**