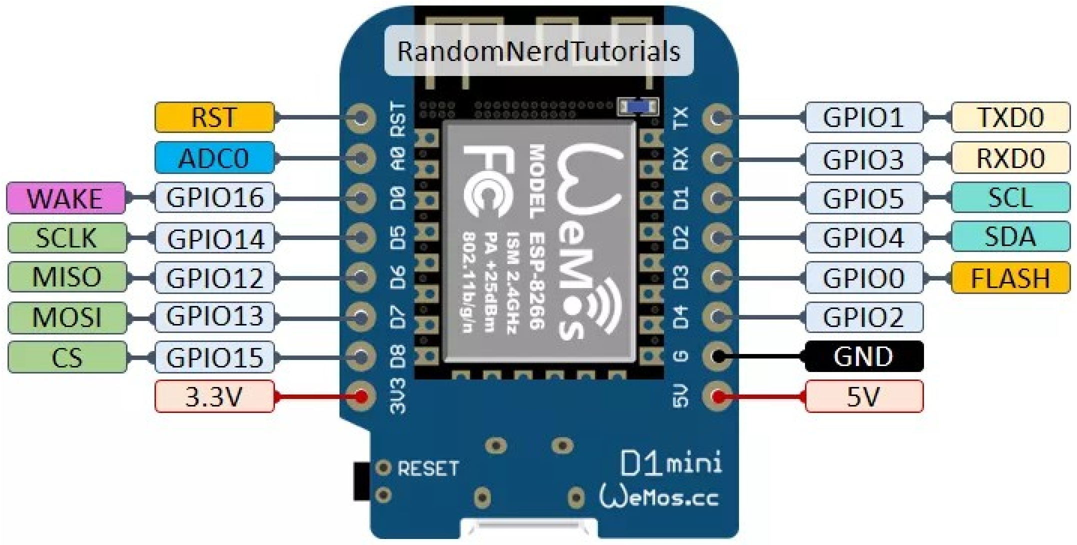
ESP8266 & Micropython: I2C

By jean-claude.feltes@education.lu

1. I2C bus and addresses

The I2C bus uses SCL (clock) and SDA (data). There must be pullup resistors (e.g. 1.5kΩ) to 3.3V. There may be several I2C slaves on on bus. Usually the addresses are chip specific, often with the possibility to change the address in a small range by setting aadress pins of the chip to High or Low.



Picture from RandomNerdTutorials

Micropython can easily scan the bus for available addresses:

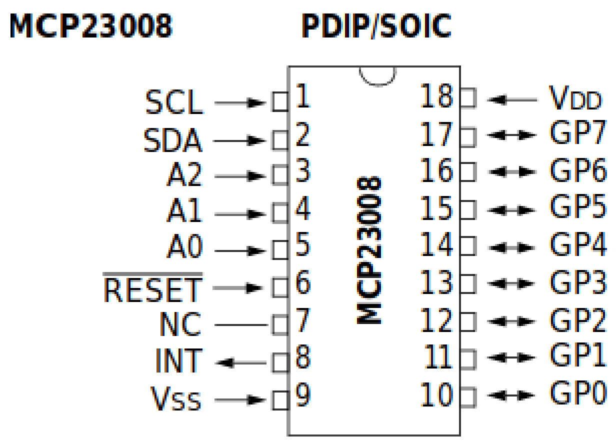
```
from machine import Pin, I2C
i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)
i2c.scan()
[32]
```

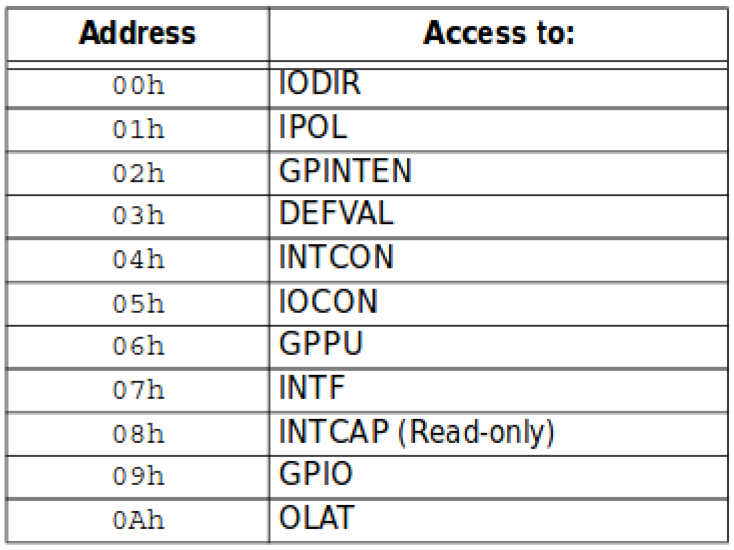
Take care: the addresses used by Micopython are 7 bit addresses.
The real addresses used by the hardware are 8 bit addresses (7 bit address shifted left one bit + RW)

Here is a simple program that lists the addresses on the bus one in a line:

```
# scan for I2C addresses:
from machine import Pin, I2C
i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)
addresses = i2c.scan()
for address in addresses:
    print(address)
```

2. Example: MCP23008 port expander





Address	Access to:
00h	IODIR
01h	IPOL
02h	GPINTEN
03h	DEFVAL
04h	INTCON
05h	IOCON
06h	GPPU
07h	INTF
08h	INTCAP (Read-only)
09h	GPIO
0Ah	OLAT

Connections:

SDA – SDA
SCL – SCL   with 1k5 pullups to 3.3V
Reset\ of MCP23008 to 3.3V

A0, A1, A2 may be connected to GND to set I2C address 32, detected by I2C scan:

```
from machine import Pin, I2C
i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)
i2c.scan()
[32]
```

Other setting for the address inputs A0, A1, A2 give other addresses from 32 to 39.

The table lists the relevant chip registers. The most important are

- IODIR defining the data direction (port as input or output)
  IODIR contains an 8 bit value, where each bit corresponds to a port pin
  0 → output, 1 → input, default is input for all: IODIR = 0xFF

- GPIO containing the data to or from the port.

3. Using a port for output

To do this we have to write 0 to the bits in IODIR that we use for output

To declare the whole port as output, we coould do this:

```
i2c.writeto(32, b'\x00\x00')
```

Here the chip has the address 32 (A0, A1, A2 connected to GND).
We write a byte array where the first \x00 is for register 0, the IODIR register, and the second \x00 is the data written to this register, in our case every bit is set 0 for all output.

After this is done, we can put data to the register number 9. These are automatically available at the port terminals.

For example:

```
i2c.writeto(33, b'\x09\xFF')     # all outputs high
i2c.writeto(33, b'\x09\x00')     # all outputs low
```

Note that before all this, we have to import the I2C library and initialise the I2C bus:

```
from machine import Pin, I2C
i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)
```

The method of writing literal byte arrays is not usable if we want to use variables.

To do this, we can use a more flexible function:

```
import struct
def write_register(address, reg, value):
    buffer = struct.pack('B', value)
    i2c.writeto_mem(address, reg, buffer)
```

Here the function write_to_mem is used, so we can use a variable for the register.
But we still have to use a byte array for the value, even if we just want to send one byte.
Here the struct library is useful: it lets us translate any data type into a byte array.

The line

```
buffer = struct.pack('B', value)
```

packs the integer "value" into a one byte array "buffer".
We must provide a formatting string, in our case 'B' for unsigned integer.

With the write_register function, it is easy for example to set bit3 of the port to output, set it high and then low again:

```
>>> write_register(33, 0, 0b11110111)      # bit 3 as output
>>> write_register(33, 9, 0b00001000)       # bit 3 set high
>>> write_register(33, 9, 0b00000000)        # bit 3 set low
```

4. Using a port for input

We can use the readfrom_mem function that is similar to the writeto_mem function:

```
def read_register(address, reg):
    data = bytearray(1)
    i2c.readfrom_mem_into(address, reg, data)      # gives for ex. bytearray(b'\x04')
    value = struct.unpack('B', data)          # gives for ex. (4,)
    return value[0]                           # gives for ex. 4
```

First we must define a one byte array “data” containing the read out data.

The readfrom_mem function returns a byte array.
This is unpacked with the struct.unpack function, returning a list of bytes “value” (in our case the list has only one element, but it is a list however). So to get our byte value, we have to take element number 0 of the list: value[0].

With this function, we can now read the value of any register on the bus:

```
>>> read_register(33, 0)
247
```

5. The beginning of a library (not only for the MCP23008)

We can put all this together into a useful I2C library:

```
''' Simple I2C library  I2C_lib_simple.py'''
from machine import Pin, I2C
import struct

i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)

def i2c_scan_bus():
    addresses = i2c.scan()
    for address in addresses:
        print(address)


def write_register(address, reg, value):
    # value is an integer 0...255
    buffer = struct.pack('B', value)
    i2c.writeto_mem(address, reg, buffer)

def read_register(address, reg):
    data = bytearray(1)
    i2c.readfrom_mem_into(address, reg, data)      # gives for ex. bytearray(b'\x04')
    value = struct.unpack('B', data)          # gives for ex. (4,)
    return value[0]                           # gives for ex. 4
```

And we can use it like this:

```
>>> from I2C_lib_simple import *
>>> read_register(33, 9)
1
```

We have used the functions for the MCP23008, but they could be used for any chip. So this simple library is quite useful for other chips also.

6. Beginning to build a library for the MCP23008

It is no great deal to gather all together in a class library:

```
from machine import Pin, I2C
import struct


class MCP23008():
    def __init__(self, address):
        self.i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)
        self.address = address

    def get_address(self):
        return self.address

    def switch_to_output_all(self):
        self.i2c.writeto(self.address, b'\x00\x00')

    def switch_to_input_all(self):
        self.i2c.writeto(self.address, b'\x00\xFF')

    def configure_port(self, iodir):
        '''sets in/out for each bit
        0 = output
        1 = input
        e.g. iodir = 0b11110000 -> P.7...P.4 = input, P.3...P.0 = output
        '''
        self.write_register(0, iodir)

    def set_all_high(self):
        self.i2c.writeto(self.address, b'\x00\x00')     # switch to output
        self.i2c.writeto(self.address, b'\x09\xFF')     # all outputs high

    def set_all_low(self):
        self.i2c.writeto(self.address, b'\x00\x00')     # switch to output
        self.i2c.writeto(self.address, b'\x09\x00')     # all outputs low

    def set_output(self, value):
        self.write_register(9, value)

    def write_register(self, reg, value):
        # value is an integer 0...255
        buffer = struct.pack('B', value)
        self.i2c.writeto_mem(self.address, reg, buffer)
    def read_register(self, reg):
        data = bytearray(1)
        self.i2c.readfrom_mem_into(self.address, reg, data)
        value = struct.unpack('B', data)
        return value[0]

    def print_all_registers(self):
        for r in range(0,11):
            value = self.read_register(r)
            print(value)
```

and so on …

Now, is it really worth while to do object oriented programming here?

Yes, as we can use the same library for any number of port extensions. And we just have to set the I2C address once. Or no, if you don't like object oriented programming and you have only one chip.
Do as you like ...

7. Using the class library

```
import MCP23008
import time
m0 = MCP23008.MCP23008(32)
m1 = MCP23008.MCP23008(33)
...
m1.switch_to_output_all()
m1.set_all_high()
time.sleep(1)
m1.set_all_low()
time.sleep(1)
m1.set_output(1)
time.sleep(1)
m1.set_output(0)
…
```

Now the question arises: is this some kind of reinventing the wheel? Surely the guys from Adafruit or other have built a library with much more functionality?
Maybe! Check the web!
But there are at least 2 reasons to do it yourself:
1. the learning effect (OK, this is a teacher speaking),
2. the fact that YOU can change the code EASILY (as you have understood it while programming) and add some cool features you need.