Easy MCP2221(A)

Reinoso Guzman (electronicayciencia@gmail.com)

Oct 23, 2022

TABLE OF CONTENTS

Easy MCP2221 is a Python module to interface with Microchip MCP2221 and MCP2221A.

Connected to the USB port, this 14 pin chip part can provide a **normal computer** with the capabilities of a **basic microcontroller**.

MCP2221's peripherals:

- 4 GPIO
- 3 channel ADC
- DAC
- I2C
- UART
- Clock Output
- USB Wake-up via Interrupt Detection.

So you can practice the basics of digital electronics, microcontrollers, and robotics using nothing more than a regular computer, a breadboard, a few parts, and Python.

Disclaimer: Some examples in this documentation show bare connections from your USB port to a breadboard. Most USB port controllers are protected against short-circuit between power and/or data lines, but some are not. I am not responsible for any damage you may cause to your computer. To be safe, always use an isolated powered USB hub for experimentation.

CHAPTER

ONE

INSTALL

EasyMCP2221 should work in Windows and Linux. Install it using `pip`.

Pip command for Linux:

```
pip install EasyMCP2221
```

Pip command for Windows:

```
py -m pip install EasyMCP2221
```

EasyMCP2221 depends on `hidapi`, which in fact needs some backend depending on OS. Sometimes this is troublesome.

If you get an error like this:

```
ImportError: Unable to load any of the following libraries:libhidapi-hidraw.so libhidapi-
↪hidraw.so.0 libhidapi-libusb.so libhidapi-libusb.so.0 libhidapi-iohidmanager.so␣
↪libhidapi-iohidmanager.so.0 libhidapi.dylib hidapi.dll libhidapi-0.dll
```

Try to install the following packages using `pip`:

- libusb
- libusb1

If that doesn't work, try manually installing libhidapi from https://github.com/libusb/hidapi/releases.

CHAPTER

TWO

EXAMPLES

First import `EasyMCP2221` and create a new *Device*.

If you can, that's great news! If you don't, check if OS is able to "see" the device or not.

```
# Just check if the MCP2221 is there.
import EasyMCP2221

try:
    mcp = EasyMCP2221.Device()

except RuntimeError:
    print("MCP2221 is not there... :(")
    exit()

print("MCP2221 is there!")
print(mcp)
```

The output should be like this:

```
MCP2221 is there!
{
        "Chip settings": {
                "Power management options": "enabled",
                "USB PID": "0x00DD",
                "USB VID": "0x04D8",
                "USB requested number of mA": 100
        },
        "Factory Serial": "01234567",
        "GP settings": {},
        "USB Manufacturer": "Microchip Technology Inc.",
        "USB Product": "MCP2221 USB-I2C/UART Combo",
        "USB Serial": "0000000000"
}
```

2.1 Basic GPIO

2.1.1 Basic digital output

Configure pin function using *set_pin_function()*, then use *GPIO_write()* to change its output.

```
# Simple example to show how to initialize class,
# set pin function and change value.
import EasyMCP2221

# Connect to device
mcp = EasyMCP2221.Device()

# Reclaim GP0 for General Purpose Input Output, as an Output.
# Default output is logical level 0.
mcp.set_pin_function(gp0 = "GPIO_OUT")

# Change it to logical 1
mcp.GPIO_write(gp0 = True)
```

2.1.2 Digital output: LED blinking

Same as before, but use *GPIO_write()* in a loop to change its output periodically.

```
# How to blink a LED connected to GP0
import sys
sys.path.append('../')

import EasyMCP2221
from time import sleep

# Connect to device
mcp = EasyMCP2221.Device()

# Reclaim GP0 for General Purpose Input Output, as an Output.
# Default output is logical level 0.
mcp.set_pin_function(gp0 = "GPIO_OUT")

while True:
    mcp.GPIO_write(gp0 = True)
    sleep(0.5)
    mcp.GPIO_write(gp0 = False)
    sleep(0.5)
```

2.1.3 Digital input: Mirror state

We introduce *GPIO_read()* this time.

In order to illustrate how to read from GPIO digital input, let's setup GP2 and GP3 to mimic the state of GP0 and GP1.

```
# GPIO output and input.
# GP0 is an output, but GP3 will be an input.
# The state of GP3 will mirror GP0.
import sys
sys.path.append('../')

import EasyMCP2221
from time import sleep

# Connect to device
mcp = EasyMCP2221.Device()

# GP0 and GP1 are inputs, GP2 and GP3 are outputs.
mcp.set_pin_function(
    gp0 = "GPIO_OUT",
    gp3 = "GPIO_IN")

while True:
    inputs = mcp.GPIO_read()
    mcp.GPIO_write(
        gp0 = inputs[3])
```

2.2 Analog signals

2.2.1 ADC basics

In this example, we setup GP1, GP2 and GP3 as analog inputs using *set_pin_function()*. Configure ADC reference with *ADC_config()* and lastly, read ADC values using *ADC_read()*.

It works better if you take off the LED and connect three potentiometers to the inputs.

Remember to **always put a 330 ohm resistor** right in series with any GP pin. That way, if you by mistake configured it as an output, the short circuit current won't exceed the 20mA.

```
# ADC input
# MCP2221 have one 10bit ADC with three channels connected to GP1, GP2 and GP3.
# The ADC is always running.
import EasyMCP2221
from time import sleep

# Connect to device
mcp = EasyMCP2221.Device()

# Use GP1, GP2 and GP3 as analog input.
mcp.set_pin_function(gp1 = "ADC", gp2 = "ADC", gp3 = "ADC")
```

(continues on next page)

(continued from previous page)

```
# Configure ADC reference
# Accepted values for ref are 'OFF', '1.024V', '2.048V', '4.096V' and 'VDD'.
mcp.ADC_config(ref="VDD")

# Read ADC values
# (adc values are always available regardless of pin function, even if output)
while True:
    values = mcp.ADC_read()

    print("ADC0: %4.1f%%    ADC1: %4.1f%%    ADC2: %4.1f%%" %
        (
        values[0] / 1024 * 100,
        values[1] / 1024 * 100,
        values[2] / 1024 * 100,
        ))

    sleep(0.1)
```

This is the console output when you move a variable resistor in GP3.

```
ADC0:  0.3%    ADC1:  0.2%    ADC2:  0.0%
ADC0:  0.3%    ADC1:  0.1%    ADC2:  0.0%
ADC0:  0.3%    ADC1:  0.2%    ADC2:  9.9%
ADC0:  0.2%    ADC1:  0.1%    ADC2: 21.7%
ADC0:  0.3%    ADC1:  0.3%    ADC2: 31.7%
ADC0:  0.2%    ADC1:  0.0%    ADC2: 38.2%
ADC0:  0.4%    ADC1:  0.3%    ADC2: 45.5%
ADC0:  0.2%    ADC1:  0.0%    ADC2: 52.3%
ADC0:  0.3%    ADC1:  0.3%    ADC2: 56.2%
ADC0:  0.1%    ADC1:  0.0%    ADC2: 58.8%
ADC0:  0.4%    ADC1:  0.2%    ADC2: 61.6%
ADC0:  0.1%    ADC1:  0.0%    ADC2: 64.6%
ADC0:  0.3%    ADC1:  0.2%    ADC2: 67.1%
ADC0:  0.2%    ADC1:  0.2%    ADC2: 70.4%
ADC0:  0.3%    ADC1:  0.1%    ADC2: 74.5%
ADC0:  0.2%    ADC1:  0.1%    ADC2: 79.2%
ADC0:  0.2%    ADC1:  0.1%    ADC2: 80.6%
```

2.2.2 Mixed signal: level meter

We will use the analog level in GP3 to set the state or three leds connected to GP0, GP1 and GP2.

```
# This could be a voltage level meter.
# GP0 and GP1 and GP2 are digital outputs.
# GP2 is analog input.
# Connect:
#   A red    LED between GP0 and positive (with a resistor).
#   A yellow LED between GP1 and positive (with a resistor).
#   A green  LED between GP2 and positive (with a resistor).
#   A potentiometer to GP3, between positive and ground.
# If potentiometer is below 25%, red led will blink.
```

(continues on next page)

(continued from previous page)

```
# Between 25% and 50%, only red will light still.
# Between 50% and 75%, red and yellow light.
# Above 75%, all three leds light.
#
# Tip: you could connect a LDR instead of a potentiometer to
# make a light level indicator.
#

import sys
sys.path.append('../')

import EasyMCP2221
from time import sleep

# Connect to device
mcp = EasyMCP2221.Device()

# GP0 and GP1 are inputs, GP2 and GP3 are outputs.
mcp.set_pin_function(
    gp0 = "GPIO_OUT",
    gp1 = "GPIO_OUT",
    gp2 = "GPIO_OUT",
    gp3 = "ADC")

mcp.ADC_config(ref="VDD")

while True:
    pot = mcp.ADC_read()[2]   # ADC channel 2 is GP3
    pot_pct = pot / 1024 * 100

    if pot_pct < 25:
        red_led_status = mcp.GPIO_read()[0]
        mcp.GPIO_write(
            gp0 = not red_led_status,
            gp1 = False,
            gp2 = False)

        sleep(0.1)

    elif 25 < pot_pct < 50:
        mcp.GPIO_write(
            gp0 = True,
            gp1 = False,
            gp2 = False)

    elif 50 < pot_pct < 75:
        mcp.GPIO_write(
            gp0 = True,
            gp1 = True,
            gp2 = False)

    elif pot_pct > 75:
```

(continues on next page)

(continued from previous page)

```
        mcp.GPIO_write(
            gp0 = True,
            gp1 = True,
            gp2 = True)
```

2.2.3 DAC: LED fading

We use *DAC_config()* and *DAC_write()* to make a LED (connected to GP3 or GP2) to fade-in and fade-out.

```
# DAC output
# MCP2221 have only 1 DAC, connected to GP2 and/or GP3.
import EasyMCP2221
from time import sleep

# Connect to device
mcp = EasyMCP2221.Device()

# Use GP2 and GP3 as DAC output.
mcp.set_pin_function(gp2 = "DAC", gp3 = "DAC")

# Configure DAC reference (max. output)
# Accepted values for ref are 'OFF', '1.024V', '2.048V', '4.096V' and 'VDD'.
mcp.DAC_config(ref="VDD")

while True:
    for v in range(0,32):
        mcp.DAC_write(v)
        sleep(0.01)

    for v in range(31,0,-1):
        mcp.DAC_write(v)
        sleep(0.01)
```

2.3 I2C bus

2.3.1 I2C bus scan

We will use *I2C_read()* to send a read command to any possible I2C address in the bus. The moment we get an acknowledge, we know there is some slave connected.

To make this example work, you need to get an EEPROM (e.g. 24LC128) and connect it properly to the SCA and SCL lines, as well as power supply.

```
# Very simple I2C scan
import EasyMCP2221

# Connect to MCP2221
```

(continues on next page)

(continued from previous page)

```
mcp = EasyMCP2221.Device()

# Optionally configure GP3 to show I2C bus activity.
mcp.set_pin_function(gp3 = "LED_I2C")

print("Searching...")

for addr in range(0, 0x80):
    try:
        mcp.I2C_read(addr)
        print("I2C slave found at address 0x%02X" % (addr))

    except RuntimeError:
        pass
```

This is my output:

```
$ python I2C_scan.py
Searching...
I2C slave found at address 0x50
```

2.3.2 Write to an EEPROM

In this example, we will use *I2C_write()* to write some string in the first memory position of an EEPROM.

```
# Simple EEPROM storage.
import EasyMCP2221

# Connect to MCP2221
mcp = EasyMCP2221.Device()

# Configure GP3 to show I2C bus activity.
mcp.set_pin_function(gp3 = "LED_I2C")

MEM_ADDR = 0x50
MEM_POS  = 0

# Take a phrase
phrase = input("Tell me a phrase: ")
# Encode into bytes using preferred encoding method
phrase_bytes = bytes(phrase, encoding = 'utf-8')

# Store in EEPROM
# Note that internal EEPROM buffer is only 64 bytes.
mcp.I2C_write(MEM_ADDR,
    MEM_POS.to_bytes(2, byteorder = 'little') +  # position to write
    bytes(phrase, encoding = 'utf-8') +          # data
    b'\0')                                       # null

print("Saved to EEPROM.")
```

Result:

```
$ python EEPROM_write.py
Tell me a phrase: This is an example.
Saved to EEPROM.
```

2.3.3 Read from an EEPROM

Same as before but reading

We seek the first position writing `0x0000`, then func:*I2C_read* 100 bytes and print until the first null.

On slower devices, the read may fail. Yo need to *I2C_cancel()* then and try again increasing func:*I2C_read* timeout parameter.

```
# Simple EEPROM storage.
import EasyMCP2221

# Connect to MCP2221
mcp = EasyMCP2221.Device()

# Configure GP3 to show I2C bus activity.
mcp.set_pin_function(gp3 = "LED_I2C")

MEM_ADDR = 0x50
MEM_POS  = 0

# Seek EEPROM to position
mcp.I2C_write(
    addr = MEM_ADDR,
    data = MEM_POS.to_bytes(2, byteorder = 'little'))

# Read max 100 bytes
data = mcp.I2C_read(
    addr = MEM_ADDR,
    size = 100)

data = data.split(b'\0')[0]
print("Phrase stored was: " + data.decode('utf-8'))
```

Output:

```
$ python EEPROM_read.py
Phrase stored was: This is an example.
```

2.4 I2C Slave helper

EasyMCP2221.I2C_Slave.I2C_Slave class allows you to interact with I2C devices in a more object-oriented way.

```
# How to use I2C Slave helper class.
# Data logger: Read 10 ADC values from a PCF8591 with 1 second interval
# and store them in an EEPROM. Then, print the stored values.
import EasyMCP2221
from time import sleep

# Connect to MCP2221
mcp = EasyMCP2221.Device()

# Create two I2C Slaves
pcf    = mcp.I2C_Slave(0x48) # 8 bit ADC
eeprom = mcp.I2C_Slave(0x50) # serial memory

# Setup analog reading (and ignore the first value)
pcf.read_register(0b00000001)

print("Storing...")
for position in range (0, 10):
    v = pcf.read()
    eeprom.write_register(position, v, reg_bytes=2)
    sleep(1)

# Dump the 10 values
v = eeprom.read_register(0x0000, 10, reg_bytes=2)
print("Data: ")
print(list(v))
```

Output:

```
$ python I2C_Slave_example.py
Storing...
Data:
[78, 78, 78, 78, 82, 102, 81, 31, 56, 77]
```

CHAPTER

THREE

FULL API REFERENCE

3.1 Device Initialization

class Device(*VID=1240*, *PID=221*, *devnum=0*)

MCP2221(A) device

Parameters

- **VID** (*int, optional*) – Vendor Id (default to 0x04D8)
- **PID** (*int, optional*) – Product Id (default to 0x00DD)
- **devnum** (*int, optional*) – Device index if multiple device found with the same PID and VID.

Raises

RuntimeError – if no device found with given VID and PID.

Example

```
>>> import EasyMCP2221
>>> mcp = EasyMCP2221.Device()
>>> print(mcp)
{
    "Chip settings": {
        "Power management options": "enabled",
        "USB PID": "0x00DD",
        "USB VID": "0x04D8",
        "USB requested number of mA": 100
    },
    "Factory Serial": "01234567",
    "GP settings": {},
    "USB Manufacturer": "Microchip Technology Inc.",
    "USB Product": "MCP2221 USB-I2C/UART Combo",
    "USB Serial": "0000000000"
}
```

3.2 Pin configuration

`set_pin_function`(*self*, *gp0=None*, *gp1=None*, *gp2=None*, *gp3=None*, *out0=False*, *out1=False*, *out2=False*, *out3=False*)

Configure pin function and, optionally, output value.

You can set multiple pins at once.

Accepted functions depends on the pin.

- For **GP0**:
  - **GPIO_IN** (*in*) : Digital input
  - **GPIO_OUT** (*out*): Digital output
  - **SSPND** (*out*): Signals when the host has entered Suspend mode
  - **LED_URX** (*out*): UART Rx LED activity output (factory default)
- For **GP1**:
  - **GPIO_IN** (*in*) : Digital input
  - **GPIO_OUT** (*out*): Digital output
  - **ADC** (*in*) : ADC Channel 1
  - **CLK_OUT** (*out*): Clock Reference Output
  - **IOC** (*in*) : External Interrupt Edge Detector
  - **LED_UTX** (*out*): UART Tx LED activity output (factory default)
- For **GP2**:
  - **GPIO_IN** (*in*) : Digital input
  - **GPIO_OUT** (*out*): Digital output
  - **ADC** (*in*) : ADC Channel 2
  - **DAC** (*out*): DAC Output 1
  - **USBCFG** (*out*): USB device-configured status (factory default)
- For **GP3**:
  - **GPIO_IN** (*in*) : Digital input
  - **GPIO_OUT** (*out*): Digital output
  - **ADC** (*in*) : ADC Channel 3
  - **DAC** (*out*): DAC Output 2
  - **LED_I2C** (*out*): USB/I2C traffic indicator (factory default)

Parameters

- **gp0** (*str, optional*) – Function for pin GP0. If None, don't alter function.
- **gp1** (*str, optional*) – Function for pin GP1. If None, don't alter function.
- **gp2** (*str, optional*) – Function for pin GP2. If None, don't alter function.
- **gp3** (*str, optional*) – Function for pin GP3. If None, don't alter function.

- **out0** (*bool, optional*) – Logic output for GP0 if configured as GPIO_OUT (default: False).
- **out1** (*bool, optional*) – Logic output for GP1 if configured as GPIO_OUT (default: False).
- **out2** (*bool, optional*) – Logic output for GP2 if configured as GPIO_OUT (default: False).
- **out3** (*bool, optional*) – Logic output for GP3 if configured as GPIO_OUT (default: False).

Raises

- **ValueError** – If invalid function for that pin is specified.
- **ValueError** – If given out value for non GPIO_OUT pin.

Examples

Set all pins at once:

```
>>> mcp.set_pin_function(
...     gp0 = "GPIO_IN",
...     gp1 = "GPIO_OUT",
...     gp2 = "ADC",
...     gp3 = "LED_I2C")
>>>
```

Change pin function at runtime:

```
>>> mcp.set_pin_function(gp1 = "GPIO_IN")
>>>
```

It is not permitted to set the output of a non GPIO_OUT pin.

```
>>> mcp.set_pin_function(
...     gp1 = "GPIO_OUT", out1 = True,
...     gp2 = "ADC", out2 = True)
Traceback (most recent call last):
...
ValueError: Pin output value can only be set if pin function is GPIO_OUT.
>>>
```

Only some functions are allowed for each pin.

```
>>> mcp.set_pin_function(gp0 = "ADC")
Traceback (most recent call last):
...
ValueError: Invalid function for GP0. Could be: GPIO_IN, GPIO_OUT, SSPND, LED_URX
>>>
```

3.3 GPIO

GPIO_read(*self*)

Read all GPIO pins logic state.

Returned values can be True, False or None if the pin is not set for GPIO operation. For an output pin, the returned status is the actual value.

Returns
4 logic values for the pins status gp0, gp1, gp2 and gp3.

Return type
tuple of bool

Example

```
>>> mcp.GPIO_read()
(None, 0, 1, None)
```

GPIO_write(*self*, *gp0=None*, *gp1=None*, *gp2=None*, *gp3=None*)

Set pin output values.

If a pin is omitted, it will preserve the value.

To change the output state of a pin, it must be assigned to GPIO_IN or GPIO_OUT (see *set_pin_function()*).

Parameters

- **gp0** (*bool, optional*) – Set GP0 logic value.
- **gp1** (*bool, optional*) – Set GP1 logic value.
- **gp2** (*bool, optional*) – Set GP2 logic value.
- **gp3** (*bool, optional*) – Set GP3 logic value.

Raises
RuntimeError – If given pin is not assigned to GPIO function.

Examples

Configure GP1 as output (defaults to False) and then set the value to logical True.

```
>>> mcp.set_pin_function(gp1 = "GPIO_OUT")
>>> mcp.GPIO_write(gp1 = True)
```

If will fail if the pin is not assigned to GPIO:

```
>>> mcp.set_pin_function(gp2 = 'DAC')
>>> mcp.GPIO_write(gp2 = False)
Traceback (most recent call last):
    ...
RuntimeError: Pin GP2 is not assigned to GPIO function.
```

3.4 ADC

ADC_config(*self*, *ref*)

Configure ADC reference voltage.

Accepted values for `ref` are "0", "1.024V", "2.048V", "4.096V" and "VDD".

Parameters
ref (*str*) – ADC reference voltage.

Raises
ValueError – if `ref` value is not valid.

Examples

```
>>> mcp.ADC_config(ref = "VDD")
```

```
>>> mcp.ADC_config("1.024V")
```

```
>>> mcp.ADC_config(ref = "5V")
Traceback (most recent call last):
...
ValueError: Accepted values for ref are 'OFF', '1.024V', '2.048V', '4.096V' and 'VDD
↪'.
```

ADC_read(*self*)

Read all Analog to Digital Converter (ADC) channels.

Analog value is always available regardless of pin function (see *set_pin_function()*). If pin is configured as output (GPIO_OUT or LED_I2C), the read value is always the output state.

ADC is 10 bits, so the minimum value is 0 and the maximum value is 1023.

Returns
Value of 3 channels (gp1, gp2, gp3).

Return type
tuple of int

Examples

All three pins configured as ADC inputs.

```
>>> mcp.ADC_config(ref = "VDD")
>>> mcp.set_pin_function(
...     gp1 = "ADC",
...     gp2 = "ADC",
...     gp3 = "ADC")
>>> mcp.ADC_read()
(185, 136, 198)
```

Reading the ADC value of a digital output gives the actual voltage in the pin. For a logic output 1 is equal to Vdd unless something is pulling that pin low (i.e. a LED).

```
>>> mcp.set_pin_function(
...     gp1 = "GPIO_OUT", out1 = True,
...     gp2 = "GPIO_OUT", out2 = False)
>>> mcp.ADC_read()
(1023, 0, 198)
```

3.5 DAC

DAC_config(*self*, *ref*, *out=0*)

Configure Digital to Analog Converter (DAC) reference.

Valid values from `ref` are "0", "1.024V", "2.048V", "4.096V" and "VDD".

MCP2221's DAC is 5 bits. So valid values for `out` are from 0 to 31.

`out` parameter is optional and defaults to 0. Use *DAC_write()* to set the DAC output value.

Parameters

- **ref** (*str*) – Reference voltage for DAC.
- **out** (*int, optional*) – value to output. Default is 0.

Raises

ValueError – if `ref` or `out` values are not valid.

Examples

```
>>> mcp.set_pin_function(gp2 = "DAC")
>>> mcp.DAC_config(ref = "4.096V")
```

```
>>> mcp.DAC_config(ref = 0)
Traceback (most recent call last):
...
ValueError: Accepted values for ref are 'OFF', '1.024V', '2.048V', '4.096V' and 'VDD
↪'.
```

DAC_write(*self*, *out*)

Set the DAC output value.

Valid `out` values are 0 to 31.

To use a GP pin as DAC, you must assign the function "DAC" (see *set_pin_function()*). MCP2221 only have 1 DAC. So if you assign to "DAC" GP2 and GP3 you will see the same output value in both.

Parameters

out (*int*) – Value to output (max. 32) referenced to DAC ref voltage.

Examples

```
>>> mcp.set_pin_function(gp2 = "DAC")
>>> mcp.DAC_config(ref = "VDD")
>>> mcp.DAC_write(31)
>>>
```

```
>>> mcp.DAC_write(32)
Traceback (most recent call last):
...
ValueError: Accepted values for out are from 0 to 31.
```

3.6 I2C

I2C_Slave(*self*, *addr*)

Create a new I2C_Slave object.

See *EasyMCP2221.I2C_Slave.I2C_Slave* for detailed information.

Parameters
addr (*int*) – Slave's I2C bus address

Returns
I2C_Slave object.

Example

```
>>> pcf    = mcp.I2C_Slave(0x48)
>>> eeprom = mcp.I2C_Slave(0x50)
>>> eeprom
EasyMCP2221's I2C slave device at bus address 0x50.
```

Note: New from v1.5.1.

I2C_write(*self*, *addr*, *data*, *kind='regular'*)

Write data to an address on I2C bus.

Maximum `data` length is 65536 bytes. If data length is 0, this function will not write anything to the bus.

Valid values for `kind` are:

- **regular**: start - data to write - stop (this is the default)
- **restart**: repeated start - data to write - stop
- **nonstop**: start - data to write

Parameters

- **addr** (*int*) – I2C slave device **base** address.
- **data** (*bytes*) – bytes to write
- **kind** (*str, optional*) – kind of transfer (see description).

Raises

- **ValueError** – if any parameter is not valid.
- **RuntimeError** – if the I2C slave didn't acknowledge.

Examples

```
>>> mcp.I2C_write(0x50, b'This is data')
>>>
```

Writing data to a non-existent device:

```
>>> mcp.I2C_write(0x60, b'This is data'))
Traceback (most recent call last):
...
RuntimeError: I2C write error: device NAK.
```

I2C_read(*self*, *addr*, *size=1*, *kind='regular'*, *timeout_ms=10*)

Read data from I2C bus.

Maximum value for `size` is 65536 bytes. If `size` is 0, only expect acknowledge from device, but do not read any bytes. Note that reading 0 bytes might cause unexpected behavior in some devices.

Valid values for `kind` are:

- **regular**: start - read data - stop
- **restart**: repeated start - read data - stop

Parameters

- **addr** (*int*) – I2C slave device **base** address.
- **size** (*int, optional*) – how many bytes to read, default 1 byte.
- **kind** (*str, optional*) – kind of transfer (see description).
- **timeout_ms** (*int, optional*) – time to wait for the data in milliseconds (default 10 ms).

Returns
data read

Return type
bytes

Raises

- **ValueError** – if any parameter is not valid.
- **RuntimeError** – if the I2C slave didn't acknowledge or the I2C engine was busy.

Examples

```
>>> mcp.I2C_read(0x50, 12)
b'This is data'
```

Solve timeout by increasing `timeout_ms` parameter:

```
>>> mcp.I2C_read(0x50, 64)
Traceback (most recent call last):
...
RuntimeError: Device did not ACK or did not send enough data. Try increasing␣
↪timeout_ms.
>>> mcp.I2C_read(0x50, 64, timeout_ms = 25)
b'This is a very long long data stream that may trigger a timeout.'
```

Hint: You can use *I2C_read()* with size 0 to check if there is any device listening with that address.

There is a device in `0x50` (EEPROM):

```
>>> mcp.I2C_read(0x50)
b''
```

No device in `0x60`:

```
>>> mcp.I2C_read(0x60)
Traceback (most recent call last):
...
RuntimeError: Device did not ACK or did not send enough data. Try increasing␣
↪timeout_ms.
```

Note: If a timeout occurs in the middle of character reading, the I2C but may stay busy. See *I2C_cancel()*.

`I2C_cancel`(*self*)

Cancel an active I2C transfer.

This command can fail in two ways.

- SCL keeps low. This is caused by:
  - Missing pull-up resistor or to high value.
  - A slave device is using clock stretching while doing an operation (e.g. writing to EEPROM).
  - Another device is using the bus.
- SDA keeps low. Caused by:
  - Missing pull-up resistor or to high value.
  - A i2c read transfer timed out while slave was sending data and now the I2C bus is locked-up. Read the Hint.

Returns

True if device is now ready to go. False if the engine is not idle.

Return type
bool

Raises

- **RuntimeError** – if I2C engine detects the **SCL** line does not go up (read description).
- **RuntimeError** – if I2C engine detects the **SDA** line does not go up (read description).

Examples

Last transfer was cancel, and engine is ready for the next operation: >>> mcp.I2C_cancel() True

```
>>> mcp.I2C_cancel()
Traceback (most recent call last):
...
RuntimeError: SCL is low. I2C bus is busy or missing pull-up resistor.
```

Hint: About the I2C bus locking-up.

Sometimes, due to a glitch or premature timeout, the master terminates the transfer. But the slave was in the middle of sending a byte. So it is expecting a few more clocks cycles to send the rest of the byte.

Since the master gave up, it will not clock the bus anymore, and so the slave won't release SDA line. The master, seeing SDA line busy, refuses to initiate any new I2C transfer. If the slave does not implement any timeout (SMB slaves do have it, but I2C ones don't), the I2C bus is locked-up forever.

MCP2221's I2C engine cannot solve this problem. You can either manually clock the bus using any GPIO line, or cycle the power supply.

Note: Do not call this function without issuing a *I2C_read()* or *I2C_write()* first. It could render I2C engine inoperative until the next reset.

```
>>> mcp.reset()
>>> mcp.I2C_is_idle()
True
>>> mcp.I2C_cancel()
False
```

Now the bus is busy until the next reset.

```
>>> mcp.I2C_speed(100000)
Traceback (most recent call last):
...
RuntimeError: I2C speed is not valid or bus is busy.
>>> mcp.I2C_cancel()
False
>>> mcp.I2C_is_idle()
False
>>> mcp.I2C_cancel()
False
```

After a reset, it will work again.

```
>>> mcp.reset()
>>> mcp.I2C_is_idle()
True
```

I2C_is_idle(*self*)

Check if the I2C engine is idle.

Returns
True if idle, False if engine is in the middle of a transfer (timeout detected).

Return type
bool

Example

```
>>> mcp.I2C_is_idle()
True
>>>
```

I2C_speed(*self*, *speed=100000*)

Set I2C bus speed.

Acceptable values for speed are between 50kHz and 400kHz.

Parameters
speed (*int*) – Bus clock frequency in Hz. Default bus speed is 100kHz.

Raises

- **ValueError** – if speed parameter is out of range.
- **RuntimeError** – if command failed (I2C engine is busy).”

Example

```
>>> mcp.I2C_speed(100000)
>>>
```

3.7 Clock output

clock_config(*self*, *duty*, *freq*)

Configure clock output frequency and Duty Cycle.

Accepted values for **duty** are: *0*, *25*, *50* and *75*.

Valid **freq** values are: *375kHz*, *750kHz*, *1.5MHz*, *3MHz*, *6MHz*, *12MHz* or *24MHz*.

To output clock signal, you also need to assign GP1 function to *CLK_OUT* (see *set_pin_function()*).

Parameters

- **duty** (*int*) – Output duty cycle in percent.
- **freq** (*str*) – Output frequency.

Raises
ValueError – if any of the parameters is not valid.

Examples

```
>>> mcp.set_pin_function(gp1 = "CLK_OUT")
>>> mcp.clock_config(50, "375kHz")
>>>
```

```
>>> mcp.clock_config(100, "375kHz")
Traceback (most recent call last):
...
ValueError: Accepted values for duty are 0, 25, 50, 75.
```

```
>>> mcp.clock_config(25, "175kHz")
Traceback (most recent call last):
...
ValueError: Freq is one of 375kHz, 750kHz, 1.5MHz, 3MHz, 6MHz, 12MHz or 24MHz
```

3.8 USB wake-up

enable_power_management(*self*, *enable=False*)

Enable or disable USB Power Management options for this device.

Set or clear Remote Wake-up Capability bit in flash configuration.

If enabled, Power Management Tab is available for this device in the Device Manager (Windows). So you can mark *"Allow this device to wake the computer"* option.

A device *reset()* (or power supply cycle) is needed in order for changes to take effect.

Parameters
enable (*bool*) – Enable or disable Power Management.

Raises

- **RuntimeError** – If write to flash command failed.
- **AssertionError** – In rare cases, when some bug might have inadvertently activated Flash protection or permanent chip lock.

Example

```
>>> mcp.enable_power_management(True)
>>> print(mcp)
...
    "Chip settings": {
        "Power management options": "enabled",
...
>>> mcp.reset()
>>>
```

wake_up_config(*self*, *edge='none'*)

Configure interruption edge.

Valid values for `edge`:

- **none**: disable interrupt detection
- **raising**: fire interruption in raising edge (i.e. when GP1 goes from Low to High).
- **falling**: fire interruption in falling edge (i.e. when GP1 goes from High to Low).
- **both**: fire interruption in both (i.e. when GP1 state changes).

In order to trigger, GP1 must be assigned to IOC function (see *set_pin_function()*).

To wake-up the computer, Power Management options must be enabled (see *enable_power_management()*). And *"Allow this device to wake the computer"* option must be set in Device Manager.

Parameters
edge (*str*) – which edge triggers the interruption (see description).

Raises
ValueError – if edge detection given.

Example

```
>>> mcp.wake_up_config("both")
>>>
```

3.9 Device reset

reset(*self*)

Reset MCP2221.

Reboot the device and load stored configuration from flash.

This operation do not reset any I2C slave devices.

3.10 Low level functions

SRAM_config(*self*, *clk_output=None*, *dac_ref=None*, *dac_value=None*, *adc_ref=None*, *int_conf=None*, *gp0=None*, *gp1=None*, *gp2=None*, *gp3=None*)

Low level SRAM configuration.

Configure Runtime GPIO pins and parameters. All arguments are optional. Apply given settings, preserve the rest.

Parameters

- **clk_output** (*int, optional*) – settings
- **dac_ref** (*int, optional*) – settings
- **dac_value** (*int, optional*) – settings
- **adc_ref** (*int, optional*) – settings

- **int_conf** (*int, optional*) – settings
- **gp0** (*int, optional*) – settings
- **gp1** (*int, optional*) – settings
- **gp2** (*int, optional*) – settings
- **gp3** (*int, optional*) – settings

Raises
RuntimeError – if command failed.

Examples

```
>>> from EasyMCP2221.Constants import *
>>> mcp.SRAM_config(gp1 = GPIO_FUNC_GPIO | GPIO_DIR_IN)
```

```
>>> mcp.SRAM_config(dac_ref = ADC_REF_VRM | ADC_VRM_2048)
```

Note: Calling this function unexpectedly resets (not preserve):

- All GPIO values set via *GPIO_write()* method.
- Reference voltage for ADC set by *ADC_config()* (not affected if ref = VDD)
- Reference voltage for DAC set by *DAC_config()* (not affected if ref = VDD)

---

send_cmd(*self*, *buf*, *sleep=0*)

Write a raw USB command to device and get the response.

Write 64 bytes to the HID interface, starting by `buf` bytes. Optionally wait `sleep` seconds. Then read 64 bytes from HID and return them as a list.

Parameters

- **buf** (*list of bytes*) – Full data to write, including command (64 bytes max).
- **sleep** (*float, optional*) – Delay (seconds) between writing the command and reading the response.

Returns
Full response data (64 bytes).

Return type
list of bytes

Example

```
>>> from EasyMCP2221.Constants import *
>>> r = mcp.send_cmd([CMD_GET_GPIO_VALUES])
[81, 0, 238, 239, 238, 239, 238, 239, 238, 239, 0, 0, 0, ... 0, 0]
```

Device.**debug_packets = False**

Print all binary commands and responses.

Type
bool

CHAPTER

FOUR

I2C SLAVE HELPER CLASS

class I2C_Slave(*mcp*, *addr*, *force=False*)

EasyMCP2221's I2C slave device.

I2C_Slave helper class allows you to interact with I2C devices in a more object-oriented way.

Usually you create new instances of this class using *EasyMCP2221.Device.I2C_Slave()* function. See *examples* section.

Parameters

- **mcp** (EasyMCP2221.Device) – MCP2221 connected to this slave
- **addr** (*int*) – Slave's I2C bus address
- **force** (*bool, optional*) – Create an I2C_Slave even if the target device does not answer. Default: False.

Raises

RuntimeError – If the device didn't acknowledge.

Examples

You should create I2C_Slave objects from the inside of an EasyMCP2221.Device:

```
>>> import EasyMCP2221
>>> mcp = EasyMCP2221.Device()
>>> eeprom = mcp.I2C_Slave(0x50)
>>> eeprom
EasyMCP2221's I2C slave device at bus address 0x50.
```

Or in a stand-alone way:

```
>>> import EasyMCP2221
>>> from EasyMCP2221 import I2C_Slave
>>> mcp = EasyMCP2221.Device()
>>> eeprom = I2C_Slave.I2C_Slave(mcp, 0x50)
```

Note: MCP2221 firmware exposes a subset of predefined I2C operations, but does not allow I2C primitives (i.e. start, stop, read + ack, read + nak, clock bus, etc.).

`is_present`()

Check if slave is present.

Perform a read operation (of 1 bytes length) to the slave address and expect acknowledge.

Returns
True if the slave answer, False if not.

Return type
bool

`read`(*length=1*)

Read from I2C slave.

See *EasyMCP2221.Device.I2C_read()*.

Parameters
length (*int*) – How many bytes to read. Default 1 byte.

Returns
list of bytes

Raises
RuntimeError – if the I2C slave didn't acknowledge or the I2C engine was busy.

`read_register`(*register*, *length=1*, *reg_bytes=1*, *reg_byteorder='big'*)

Read from a specific register, position or command.

Sequence:

- Start
- Send device I2C address + R/W bit 0
- Send register byte, memory position or command
- Repeated start
- Send device I2C address + R/W bit 1
- Read `length` bytes
- Stop

See *EasyMCP2221.Device.I2C_read()* for more information.

Parameters

- **register** (*int*) – Register to read, memory position or command.
- **length** (*int, optional*) – How many bytes is the answer to read (default read 1 byte).
- **reg_bytes** (*int, optional*) – How many bytes is the register, position or command to send (default 1 byte).
- **reg_byteorder** (*str, optional*) – Byte order of the register address. *'little'* or *'big'*. Default 'big'.

Returns
list of bytes

Examples

Read from a regular i2c device, register 0x0D:

```
>>> bme.read_register(0x0D)
>>> b'ÿ'
```

Read 10 bytes from I2C EEPROM (2 bytes memory position):

```
>>> eeprom.read_register(2000, 25, reg_bytes=2)
>>> b'en muchas partes hallaba '
```

write(*data*)

Write to I2C slave.

See *EasyMCP2221.Device.I2C_write()* for more information.

Parameters
data (*bytes*) – Data to write. Bytes, int from 0 to 255, or list of ints from 0 to 255.

Raises
RuntimeError – if the I2C slave didn't acknowledge or the I2C engine was busy.

write_register(*register*, *data*, *reg_bytes=1*, *reg_byteorder='big'*)

Write to a specific register, position or command.

Sequence:

- Start
- Send device I2C address + R/W bit 0
- Send register byte, memory position or command
- Repeated start
- Send device I2C address + R/W bit 0
- Write `data`
- Stop

See *EasyMCP2221.Device.I2C_write()* for more information.

Parameters

- **register** (*int*) – Register to read, memory position or command.
- **data** (*bytes*) – Data to write. Bytes, int from 0 to 255, or list of ints from 0 to 255.
- **reg_bytes** (*int, optional*) – How many bytes is the register, position or command to send (default 1 byte).
- **reg_byteorder** (*str, optional*) – Byte order of the register address. *'little'* or *'big'*. Default 'big'.

Examples

Set PCF8591's DAC output to 255. Command 0bx1xxxxxx.

```
>>> pcf.write_register(0b01000000, 255)
```

Write a stream of bytes to an EEPROM at position 0x1A00 (2 bytes memory position):

```
>>> eeprom.write_register(0x1A00, b'Testing 123...', reg_bytes=2)
>>> eeprom.read_register(0x1A00, 14, reg_bytes=2)
b'Testing 123...'
```

CHAPTER

FIVE

LIMITATIONS AND BUGS

5.1 Limitations

USB polling rate for this device is 250Hz. That means:

- Maximum GPIO output frequency using *GPIO_write()*: 250Hz
- Maximum GPIO output frequency using *set_pin_function()*: 120Hz aprox.
- Maximum GPIO input frequency for *GPIO_read()*: 250Hz
- This also affects to ADC reading rate, DAC updating, and so on.

The ADC seems to be always connected. So leakage current for GP1, GP2 and GP3 is greater than for GP0. Think of it as a very weak *pull-down* resistor on these pins.

5.2 Bugs

None reported.

Bug tracking system: https://github.com/electronicayciencia/EasyMCP2221/issues

CHAPTER

SIX

HISTORY

6.1 V1.5

6.1.1 V1.5.1

Add I2C Slave helper class.

6.1.2 V1.5.0

First EasyMCP2221 version.

6.2 Older releases

This project was initially a fork of PyMCP2221A library by Yuta KItagami (https://github.com/nonNoise/PyMCP2221A).

I did a few changes, then a few more, until I ended up rewriting almost all the code. Since the API is no longer compatible with PyMCP2221A, I decided to create a new package.

Tags v1.4 and earlier are from PyMCP2221A.

CHAPTER

SEVEN

LINKS

- Source GitHub repository:

  https://github.com/electronicayciencia/EasyMCP2221

- MCP2221 - USB 2.0 to I2C/UART Protocol Converter with GPIO - Microchip Inc.

  https://www.microchip.com/en-us/product/MCP2221

- 24LC128 - 128Kb I2C compatible 2-wire Serial EEPROM - Microchip Inc.

  https://www.microchip.com/en-us/product/24LC128

- CircuitPython Libraries on any Computer with MCP2221. (A similar project but using Adafruit's MCP2221A breakout board instead of a bare chip)

  https://learn.adafruit.com/circuitpython-libraries-on-any-computer-with-mcp2221

  https://github.com/adafruit/Adafruit_Blinka/blob/main/src/adafruit_blinka/microcontroller/mcp2221/mcp2221.py

- Another breakout board: Artekit AK-MCP2221

  https://www.artekit.eu/doc/guides/ak-mcp2221/

INDEX