Easy MCP2221(A)

Reinoso Guzman (electronicayciencia@gmail.com)

Mar 03, 2023

TABLE OF CONTENTS

1 Disclaimer 3

2 MIT License 5

Index 55

EasyMCP2221 is a **Python** module to interface with Microchip MCP2221 and MCP2221A focused on ease of use.

The MCP2221 is a fully integrated USB-to-UART/I2C serial converter with four GP pins providing miscellaneous functionalities.

MCP2221's peripherals:

- 4 General Purpose Input/Output (GPIO) pins
- 3 channel 10 bit ADC
- One 5 bit DAC
- I2C
- UART
- Clock Output with PWM
- USB Wake-up via Interrupt Detection.

With this chip and this library you can practice the basics of digital electronics, microcontrollers, and robotics with a regular computer and regular Python. See *Examples*.

This is MCP2221 and **MCP2221A pinout**:

CHAPTER

ONE

DISCLAIMER

I am not related to Microchip Inc. in any way. This library is unofficial and for personal use only.

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.

Many thanks to Microchip for providing free samples of MCP2221A, and for openly publishing the datasheet and documentation used to write this library.

CHAPTER

TWO

MIT LICENSE

2.1 Install / troubleshooting

2.1.1 Regular installation via PIP

Pip command for Linux:

```
$ pip install EasyMCP2221
```

On Linux, additional steps may be required. See *Troubleshooting* section below.

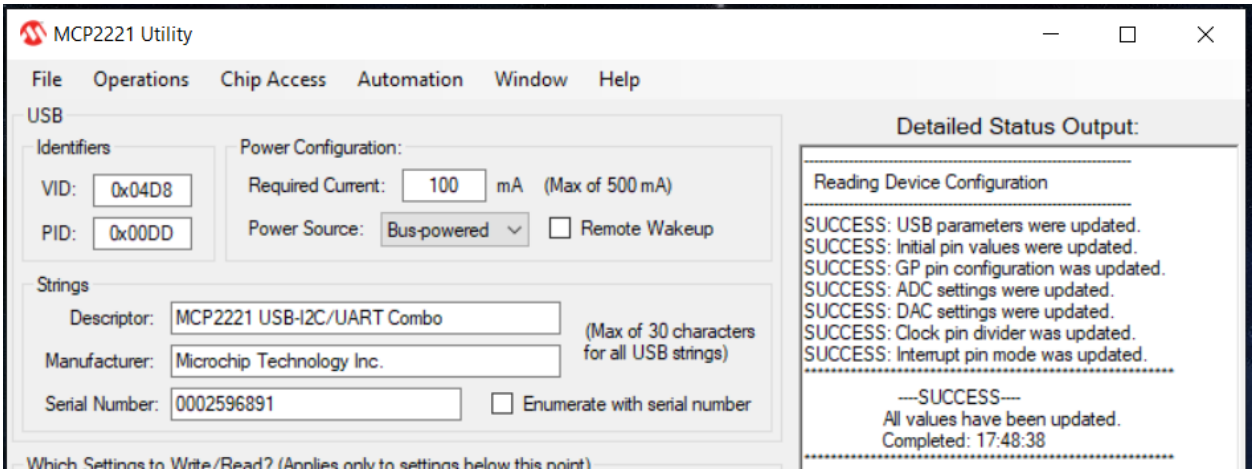
Pip command for Windows:

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

2.1.2 Troubleshooting

Device not found

First step, try to use any of the Microchip's official tools to verify that everything is working fine.



On Linux, use `lsusb` to find any device with ID `04D8:xxxx`:

```
$ lsusb
Bus 001 Device 004: ID 04d8:00dd Microchip Technology, Inc.
...
```

Hidapi backend error

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.

Sometimes, yo need to manually copy `libusb-1.0.dll` to `C:\Windows\System32`. It used to be in `C:\Users\[username]\AppData\Local\Programs\Python\Python39\Lib\site-packages\libusb\_platform\_windows\x64\libusb-1.0.dll` or similar path.

Open failed for non-root users (Linux)

On **Linux**, only root can open these devices. Trying to run the software without privileges will raise the following error:

```
$ python3 pruebas.py
Traceback (most recent call last):
  File "pruebas.py", line 7, in <module>
    mcp = EasyMCP2221.Device(trace_packets = False)
  File "/home/pi/EasyMCP2221/EasyMCP2221/MCP2221.py", line 82, in __init__
    self.hidhandler.open_path(hid.enumerate(VID, PID)[devnum]["path"])
  File "hid.pyx", line 142, in hid.device.open_path
OSError: open failed
```

To change that, you need to add a udev rule. Create a new file under `/etc/udev/rules.d/99-mcp.rules` with this content:

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="04d8", MODE="0666", GROUP="plugdev"
```

Delay at the end of script (Linux)

If you experience delays on script startup or exit, use `lsmod` to check for conflicting drivers.

```
# lsmod | grep hid
hid_mcp2221            20480  1
hid_generic            16384  0
usbhid                 57344  0
hid                   139264  3 usbhid,hid_generic,hid_mcp2221
```

This library may conflict with `hid_mcp2221` kernel module.

To blacklist this module, create a file named `/etc/modprobe.d/blacklist-mcp2221.conf` with this content:

```
blacklist hid_mcp2221
```

Run `rmmod hid_mcp2221` to unload the module.

2.1.3 Local installation and testing

You may want to install this library from a cloned GitHub repository, usually for testing or development purposes.

First create and activate a new virtual environment. Update pip if needed.

```
> python -m venv init easymcp_dev
> cd easymcp_dev
> Scripts\activate
> python -m pip install --upgrade pip
```

Then, clone the home repository inside that virtual environment and perform the installation in *editable* (`-e`) mode.

```
$ git clone https://github.com/electronicayciencia/EasyMCP2221

$ pip install -e EasyMCP2221
```

If you get this error: *"File "setup.py" not found. Directory cannot be installed in editable mode"*, update PIP.

```
> python -m pip install --upgrade pip
```

If you get this one: *"EasyMCP2221 does not appear to be a Python project: neither 'setup.py' nor 'pyproject.toml' found."*, please check working directory. You must be in the root of the cloned GitHub repository.

Local documentation

This is an optional step. To compile documentation locally you will need `sphinx` and `RTD theme`.

```
pip install -U sphinx
pip install -U sphinx_rtd_theme
```

Compilation:

```
cd docs
make html
```

Main HTML file is *EasyMCP2221/docs/build/html/index.html*.

Testing

There is a test suite to check ADC, DAC, I2C, and some other features like start-up and persistence after a reset.

In order to pass the tests, you need a working MCP2221 or MCP2221A and a serial EEPROM 24LC128 or bigger. Use the following schematic:

GP0 and **GP1** are used to test I2C in several scenarios. **GP3** is used as a DAC. **GP2**, connected to a simple RC low pass filter, is used as an ADC to test different voltaje references.

Run all tests:

```
$ python -m unittest
```

Run specific test suite, verbose and fail-fast:

```
$ python -m unittest test.test_gpio -fv
```

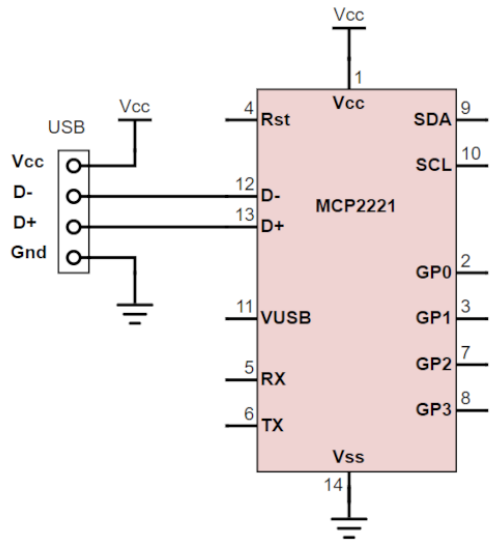
2.2 Examples

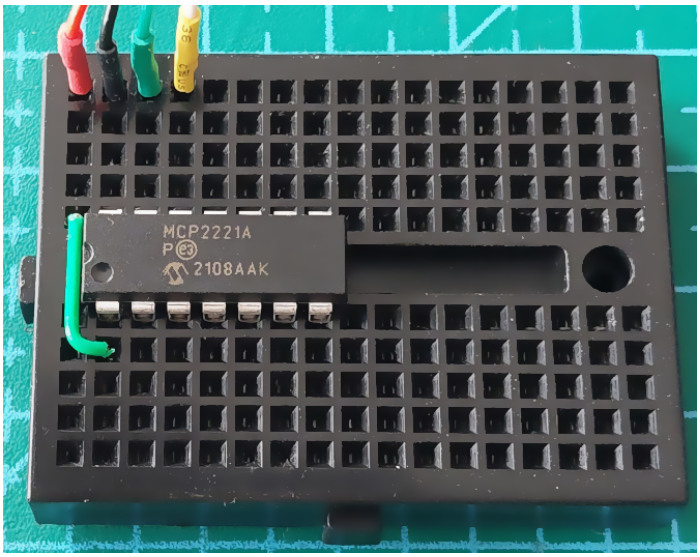
2.2.1 Getting started

This is the minimum part layout.

Although this is **not a recommended design**, it should work fine. See the MCP2221 data sheet for more information.

Import `EasyMCP2221` module and try to create a new *`Device`* object with default parameters.






```
>>> 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"
}
```

In case of error, make sure MCP2221A is properly connected. Use Microchip's tool to find the device. Also read the troubleshooting section in *Install / troubleshooting*.

2.2.2 Basic GPIO

Configure pin function using *set_pin_function()* to GPIO_IN or GPIO_OUT. Then use *GPIO_write()* to change its output. Or *GPIO_read()* to read the status.

Digital output: LED blinking

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

Schematic:

Code:

```
# How to blink a LED connected to GP2
import EasyMCP2221
from time import sleep

# Connect to the device
mcp = EasyMCP2221.Device()

# Reclaim GP2 for General Purpose Input Output, as an Output.
mcp.set_pin_function(gp2 = "GPIO_OUT")

while True:
    mcp.GPIO_write(gp2 = True)
    sleep(0.5)
    mcp.GPIO_write(gp2 = False)
    sleep(0.5)
```

Result:

Digital input: Mirror state

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 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])
```

2.2.3 Analog signals

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")

# 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%
```

(continues on next page)

(continued from previous page)

```
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%
```

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.
# 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 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,
```

(continues on next page)

(continued from previous page)

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

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 with a triangular wave.

```
# DAC output
# MCP2221 only have 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(30,0,-1):
        mcp.DAC_write(v)
```

(continues on next page)

(continued from previous page)

```
        #sleep(0.01)
```

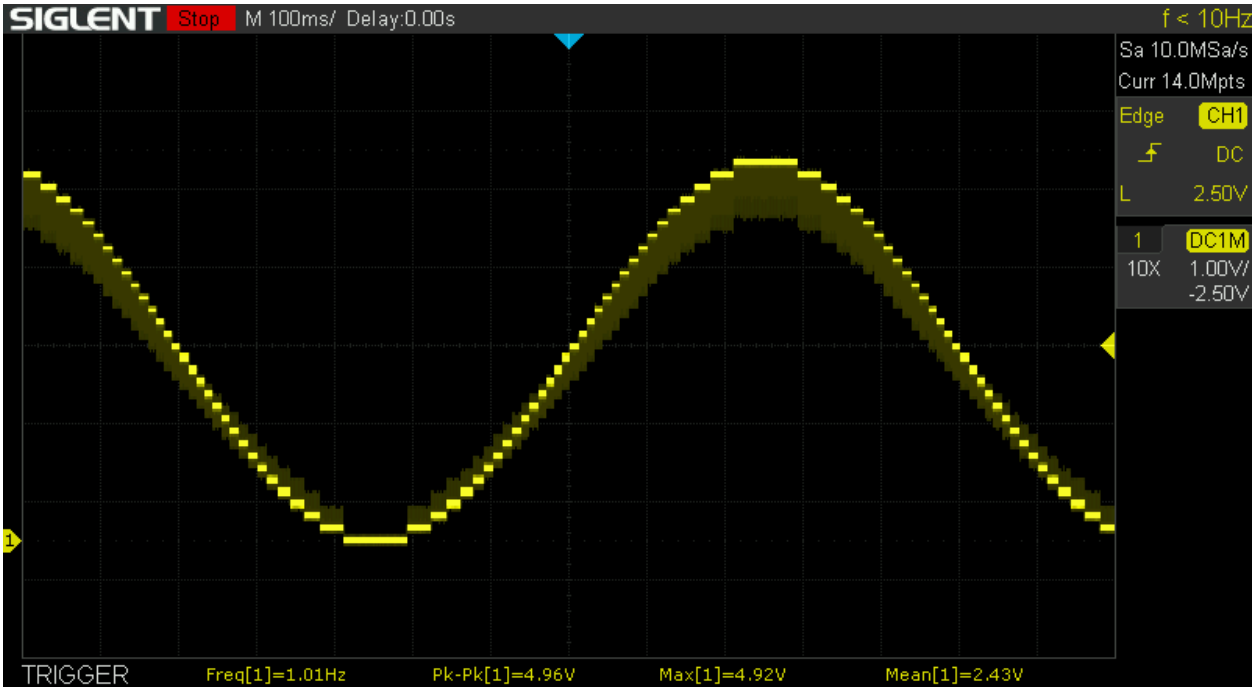
2.2.4 Advanced analog

Sinusoidal generator

In the following example, we will use DAC to generate a `sin` waveform with a period of 1 second.

DAC's maximum update rate is 500Hz, one sample every 2ms on average. It really depends on the load of the host and USB bus controller.

DAC's resolution is only 5 bit. That means 32 different values.



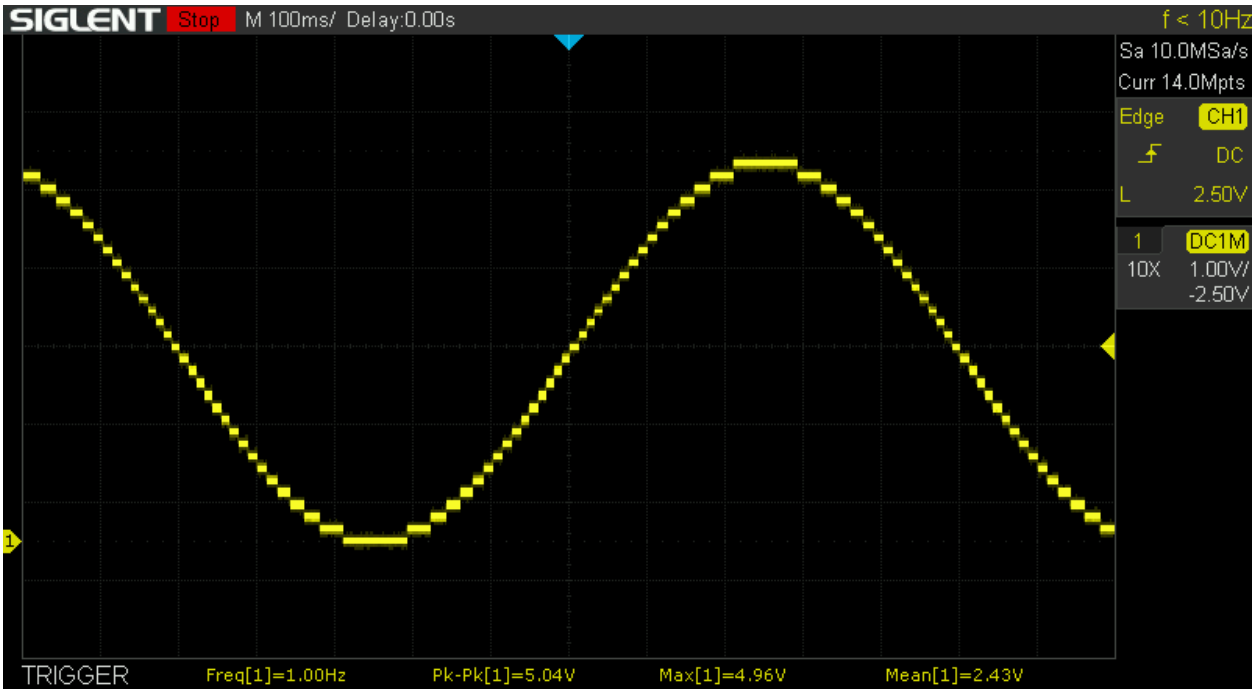
Noise comes from USB traffic and it is in kHz region. Since ADC output frequency is much lower, it can be greatly reduced with a simple RC low pass filter.

Notice the usage of `time.perf_counter()` instead of `sleep` to get a more or less constant rate in a multitask operating system.

```
# DAC output, advanced example.
# Generate SIN signal using a recurrence relation to avoid calculate sin(x) in the main␣
↪loop.
import EasyMCP2221
import time
from math import sqrt, cos, pi

# Output freq
sample_rate = 500 # Hz (unstable above 500Hz)
freq        = 1   # Hz
```

(continues on next page)



(continued from previous page)

```
# Configure device pins and DAC reference.
# MCP2221 have only 1 DAC, connected to GP2 and/or GP3.
mcp = EasyMCP2221.Device()
mcp.set_pin_function(gp2 = "DAC", gp3 = "DAC")
mcp.DAC_config(ref="VDD")

# Initial values
W             = cos(2*pi*freq/sample_rate)
last_s        = sqrt(1-W**2)  # y_n-1  (y1)
before_last_s = 0             # y_n-2  (y0)

# No trigonometric function in the main loop
while True:
    # set-up next sample time before doing anything else
    next_sample = time.perf_counter() + 1/sample_rate

    # Calculate next output value and write it to DAC
    s = 2*W*last_s - before_last_s    # s between -1 and 1
    out = (s + 1) / 2   # out between 0 and 1 now
    out = out * 31      # 5 bit DAC, 0 to 31
    out = round(out)    # integer
    mcp.DAC_write(out)

    # Update recurrence values
    (before_last_s, last_s) = (last_s, s)

    # Warn if we can't keep up with the sample rate!
    if time.perf_counter() > next_sample:
```

(continues on next page)

(continued from previous page)

```
        print("Undersampling!")

    # Wait fixed delay for next sample (do not use sleep)
    while time.perf_counter() < next_sample:
        pass
```

Capacitor charge

A GPIO output can be used to charge or discharge a capacitor through a resistor while we are sampling ADC values at regular intervals:

Program:

```
# Plotter for capacitor change/discharge
import EasyMCP2221
import time
import matplotlib.pyplot as plt
import numpy as np

capture_time = 1
Vdd = 5

# Configure device pins
mcp = EasyMCP2221.Device()
mcp.ADC_config()
mcp.set_pin_function(gp2 = "GPIO_OUT", gp3 = "ADC")

V = []
T = []

print("Initial discharge on course. Press enter to start charging.")
mcp.GPIO_write(gp2 = False)

input()
print("Charging...")
mcp.GPIO_write(gp2 = True)

start = time.perf_counter()

while time.perf_counter() - start <= capture_time:

    t = time.perf_counter()
    (_, _, V3) = mcp.ADC_read()

    # 10 bit, 5V ref
    V3 = V3 / 1024 * Vdd

    T.append(t - start)
    V.append(V3)
```

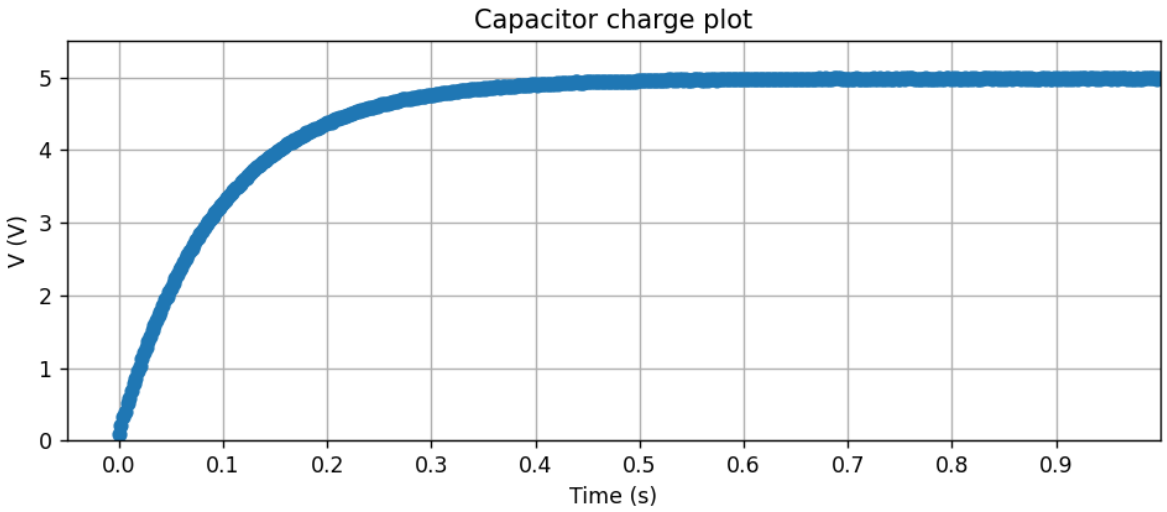
(continues on next page)

(continued from previous page)

```
mcp.GPIO_write(gp2 = False)


plt.plot(T, V, 'o-')
plt.axis([-0.05, capture_time, 0, Vdd + 0.5])
plt.xticks(np.arange(0,capture_time,0.1))
plt.xlabel("Time (s)")
plt.ylabel("V (V)")
plt.title("Capacitor charge plot")
plt.grid()
plt.show()
```

This will produce the classic capacitor charge curve:



LED V/I plotter

We can read the ADC values while we are changing the DAC output to characterize some part.

Note that the DAC output impedance is 5k (according to the datasheet), so you can't draw much current from it.

The breadboard connections are pretty straightforward:

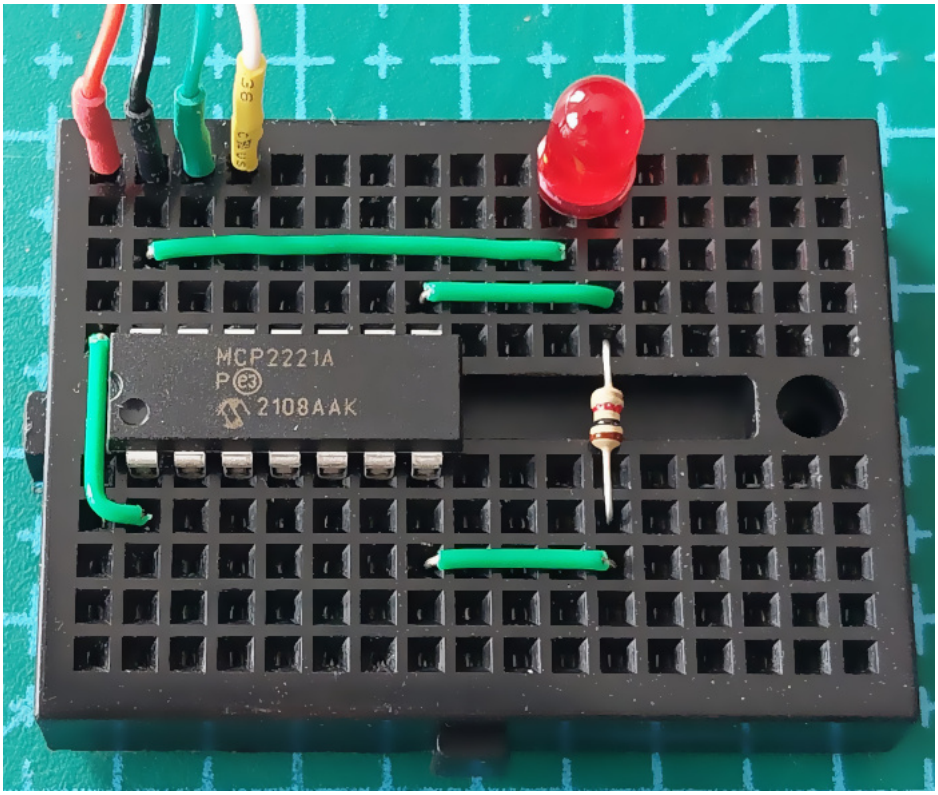
Program:

```
# V/I plotter DAC/ADC example.
import EasyMCP2221
from time import sleep

import matplotlib.pyplot as plt

# Configure device pins ADC and DAC reference.
```

(continues on next page)



(continued from previous page)

```
# DAC output impedance is about 5k according to datasheet
# so measurements could be inaccurate as the current increases.
mcp = EasyMCP2221.Device()
mcp.set_pin_function(gp2 = "DAC", gp3 = "ADC")
mcp.DAC_config()
mcp.ADC_config()

R = 1000

V = 32 * [0]
I = 32 * [0]

for step in range(0,32):
    mcp.DAC_write(step)
    (_, V2, V3) = mcp.ADC_read()

    # 10 bit, 5V ref
    V2 = V2 / 1024 * 5
    V3 = V3 / 1024 * 5

    # I = V/R
    I_r = (V2 - V3) / R

    V[step] = V2
    I[step] = I_r * 1000 # mA
```

(continues on next page)

(continued from previous page)

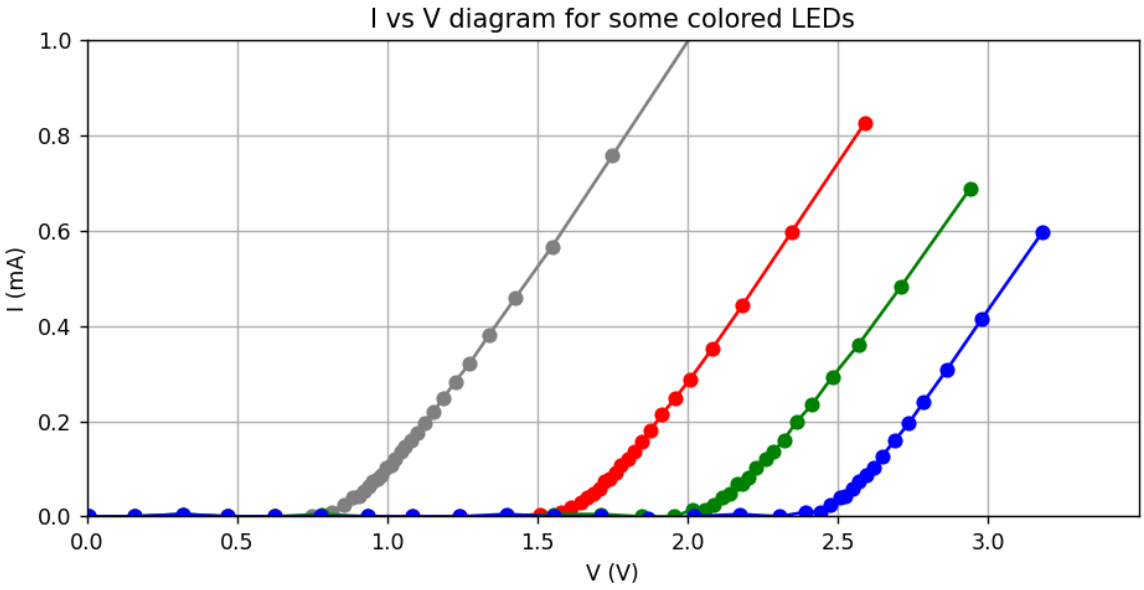
```
    print("Step:", step+1, "/ 32")

    sleep(0.05)


mcp.DAC_write(0)

plt.plot(V, I, 'o-')
plt.axis([0,5,0,1])
plt.xlabel("V (V)")
plt.ylabel("I (mA)")
plt.title("I vs V diagram")
plt.grid()
plt.show()
```

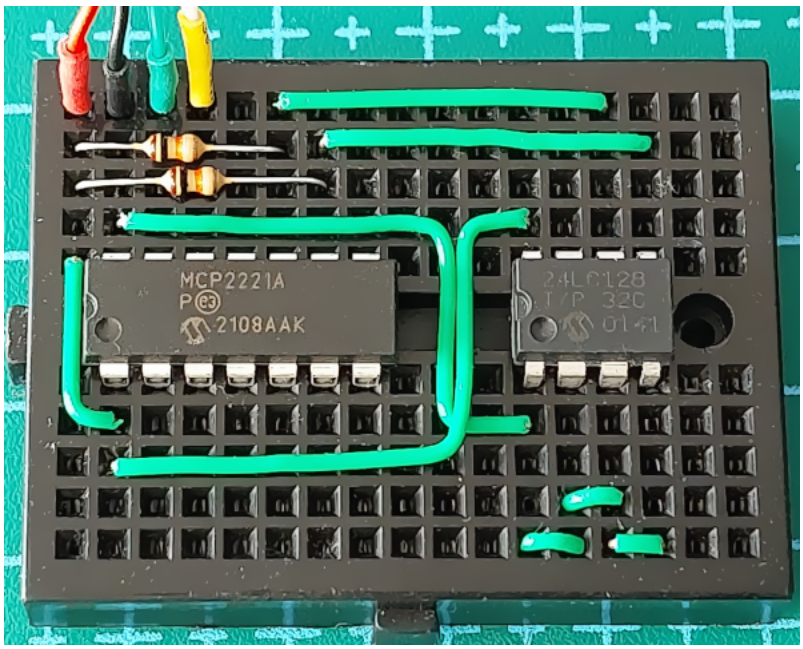
This is the output for an infrared, red, green and blue LEDs.



2.2.5 I2C bus

To make these examples 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.

This is it in the breadboard. Don't forget to connect *WP* pin to either `Vcc` or `Gnd`.



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.

```
# Very simple I2C scan
import EasyMCP2221

# Connect to MCP2221
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 EasyMCP2221.exceptions.NotAckError:
        pass
```

This is my output:

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

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.
```

Read from an EEPROM

Same as before but reading

We seek the first position writing **0x0000**, then *I2C_read()* 100 bytes and print until the first null.

```
# Simple EEPROM reading.
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
```

(continues on next page)

(continued from previous page)

```
# 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.
```

2.2.6 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]
```

2.3 Full API reference

2.3.1 Device Initialization

class Device(*VID=None*, *PID=None*, *devnum=None*, *trace_packets=None*)

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. Default is first device (index 0).
- **trace_packets** (*bool, optional*) – For debug only. See *trace_packets*.

Raises

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

Example

```
>>> import EasyMCP2221
>>> mcp = EasyMCP2221.Device()
>>> print(mcp)
{
    "Chip settings": {
        "Interrupt detection edge": "both",
        "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"
}
```

2.3.2 Pin configuration

`set_pin_function`(*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.

GP0 functions:

- **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)

GP1 functions:

- **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)

GP2 functions:

- **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)

GP3 functions:

- **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", out1 = True,
...     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
>>>
```

Hint: Pin assignments are active until reset or power cycle. Use *save_config()* to make this configuration the default at next start.

---

save_config()

Write current status (pin assignments, GPIO output values, DAC reference and value, ADC reference, etc.) to flash memory.

You can save a new configuration as many times as you wish. That will be the default state at power up.

Raises

- **RuntimeError** – if command failed.
- **AssertionError** – if an accidental flash protection attempt was prevented.

Example

Set all GPIO pins as digital inputs (high impedance state) at start-up to prevent short circuits while breadboarding.

```
>>> mcp.set_pin_function(
...     gp0 = "GPIO_IN",
...     gp1 = "GPIO_IN",
...     gp2 = "GPIO_IN",
...     gp3 = "GPIO_IN")
>>> mcp.DAC_config(ref = "OFF")
>>> mcp.ADC_config(ref = "VDD")
>>> mcp.save_config()
```

2.3.3 GPIO

GPIO_read()

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(*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.
```

2.3.4 ADC

ADC_read()

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)
```

ADC_config(*ref='VDD'*)

Configure ADC reference voltage.

ref values:

- "OFF"
- "1.024V"
- "2.048V"
- "4.096V"
- "VDD"

Parameters
ref (*str, optional*) – ADC reference value. Default to supply voltage (Vdd).

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

Examples

```
>>> mcp.ADC_config()
```

```
>>> 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
↪'.
```

Hint: ADC configuration is saved when you call *save_config()* and reloaded at power-up. You only need to call this function if you want to change it.

2.3.5 DAC

DAC_write(*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.
```

DAC_config(*ref='VDD'*, *out=None*)

Configure Digital to Analog Converter (DAC) reference.

ref values:

- "OFF"
- "1.024V"
- "2.048V"
- "4.096V"
- "VDD"

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

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

Parameters

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

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
↪'.
```

Hint: DAC configuration is saved when you call *save_config()* and reloaded at power-up. You only need to call this function if you want to change it.

2.3.6 I2C

I2C_Slave(*addr*, *force=False*, *speed=100000*)

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.
```

I2C_write(*addr*, *data*, *kind='regular'*, *timeout_ms=20*)

Write data to an address on I2C bus.

Valid values for **kind** are:

regular
It will send **start**, *data*, **stop** (this is the default)

restart
It will send **repeated start**, *data*, **stop**

nonstop
It will send **start**, data to write, (no stop). Please note that you must use 'restart' mode to read or write after a *nonstop* write.

Parameters

- **addr** (*int*) – I2C slave device **base** address.
- **data** (*bytes*) – bytes to write. Maximum length is 65535 bytes, minimum is 1.
- **kind** (*str, optional*) – kind of transfer (see description).
- **timeout_ms** (*int, optional*) – maximum time to write data chunk in milliseconds (default 20 ms). Note this time applies for each 60 bytes chunk. The whole write operation may take much longer.

Raises

- **ValueError** – if any parameter is not valid.
- *NotAckError* – if the I2C slave didn't acknowledge.
- *TimeoutError* – if the writing timeout is exceeded.
- *LowSDAError* – if I2C engine detects the **SCL** line does not go up (read exception description).
- *LowSCLError* – if I2C engine detects the **SDA** line does not go up (read exception description).

- **RuntimeError** – if some other error occurs.

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):
...
EasyMCP2221.exceptions.NotAckError: Device did not ACK.
```

Note: MCP2221 writes data in 60-byte chunks.

The default timeout of 20 ms is twice the time required to send 60 bytes at the minimum supported rate (47 kHz).

MCP2221's internal I2C engine has additional timeout controls.

I2C_read(*addr*, *size=1*, *kind='regular'*, *timeout_ms=20*)

Read data from I2C bus.

Valid values for `kind` are:

regular
: It will send **start**, *data*, **stop** (this is the default)

restart
: It will send **repeated start**, *data*, **stop**

Parameters

- **addr** (*int*) – I2C slave device **base** address.
- **size** (*int, optional*) – how many bytes to read. Maximum is 65535 bytes. Minimum is 1 byte.
- **kind** (*str, optional*) – kind of transfer (see description).
- **timeout_ms** (*int, optional*) – time to wait for the data in milliseconds (default 20 ms). Note this time applies for each 60 bytes chunk. The whole read operation may take much longer.

Returns
: data read

Return type
: bytes

Raises

- **ValueError** – if any parameter is not valid.
- ***NotAckError*** – if the I2C slave didn't acknowledge.
- ***TimeoutError*** – if the writing timeout is exceeded.

- *LowSDAError* – if I2C engine detects the **SCL** line does not go up (read exception description).
- *LowSCLError* – if I2C engine detects the **SDA** line does not go up (read exception description).
- **RuntimeError** – if some other error occurs.

Examples

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

Write then Read without releasing the bus:

```
>>> mcp.I2C_write(0x50, position, 'nonstop')
>>> mcp.I2C_read(0x50, length, 'restart')
b'En un lugar de la Mancha...'
```

Hint: You can use *I2C_read()* with size 1 to check if there is any device listening with that address.

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

```
>>> mcp.I2C_read(0x50)
b'1'
```

No device in `0x60`:

```
>>> mcp.I2C_read(0x60)
Traceback (most recent call last):
...
EasyMCP2221.exceptions.NotAckError: Device did not ACK.
```

Note: MCP2221 reads data in 60-byte chunks.

The default timeout of 20 ms is twice the time required to receive 60 bytes at the minimum supported rate (47 kHz). If a timeout or other error occurs in the middle of character reading, the I2C may get locked. See *LowSDAError*.

I2C_speed(*speed=100000*)

Set I2C bus speed.

Acceptable values for speed are between 47kHz 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)
>>>
```

Note: Between 47kHz and 400kHz is the recommended value. The minimum value is actually 46693, which corresponds to a clock of approximately 46.5kHz. And the maximum is 6000000, that generates about 522kHz clock.

`I2C_cancel()`

Try to cancel an active I2C read or write command.

Warning: Deprecated.

There is no need for this method anymore. The bus is managed automatically. You can use *_i2c_release()* function to manually cancel a transfer.

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

Return type
bool

Raises

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

Examples

Last transfer was cancel, and engine is ready for the next operation:

```
>>> mcp.I2C_cancel()
True
```

Last transfer failed, and cancel failed too because I2C bus seems busy:

```
>>> mcp.I2C_cancel()
Traceback (most recent call last):
...
EasyMCP2221.exceptions.LowSCLError: SCL is low. I2C bus is busy or missing pull-up␣
↪resistor.
```

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()
```

(continues on next page)

(continued from previous page)

```
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()`

Check if the I2C engine is idle.

> **Warning: Deprecated.**
>
> There is no need for this method anymore. The bus is managed automatically. You can use *_i2c_status()* function to know the I2C internal status details.

Returns
True if idle, False if engine is in the middle of a transfer or busy.

Return type
bool

Raises

- ***LowSDAError*** – if **SCL** line is down (read exception description).
- ***LowSCLError*** – if **SDA** line is down (read exception description).

Example

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

2.3.7 Clock output

clock_config(*duty*, *freq*)

Configure clock output frequency and Duty Cycle.

duty values:

- 0
- 25
- 50
- 75

freq values:

- "375kHz"
- "750kHz"
- "1.5MHz"
- "3MHz"
- "6MHz"
- "12MHz"
- "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
```

2.3.8 USB wake-up

enable_power_management(*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(*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.

Remember to call *save_config()* to persist this configuration when the chip resets

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

Raises
ValueError – if edge detection given.

Example

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

2.3.9 Device reset

reset()

Reset MCP2221.

Reboot the device and load stored configuration from flash.

This operation do not reset any I2C slave devices.

Note: The host needs to re-enumerate the device after a reset command. There is a 5 seconds timeout to do that.

2.3.10 Low level and debug

SRAM_config(*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 to change GPIO when DAC is active and DAC reference is not Vdd will create a 2ms gap in DAC output.

---

send_cmd(*buf*)

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

Write 64 bytes to the HID interface, starting by `buf` bytes. Then read 64 bytes from HID and return them as a list. In case of failure (USB read/write or command error) it will retry. To prevent this, set *cmd_retries* to zero.

Parameters
buf (*list of bytes*) – Full data to write, including command (64 bytes max).

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]
```

See also:

Class variables *cmd_retries*, *debug_messages* and *trace_packets*.

Hint: The response does not wait until the actual command execution is finished. Instead, it is generated right after the device receives the command. So an error response might indicate:

- the most recent command is not valid
- the previous command finished with an error condition (case of I2C write).

_i2c_release()

Try to make the I2C bus ready for the next operation.

This is a private method, the **API can change** without previous notice.

If there is an active transfer, cancel it. Try multiple times.

Determine if the bus is ready monitoring SDA and SCL lines.

Raises

- *LowSDAError* – if **SCL** line is down (read exception description).

- *LowSCLError* – if **SDA** line is down (read exception description).
- **RuntimeError** – if multiple cancel attempts did not work. Undetermined cause.

Note: Calling *Cancel* command on an uninitialized I2C engine can make it crash in 0x62 status until next reset. This function uses *_i2c_status()* heuristics to determine if it can issue a Cancel now or not.

_i2c_status()

Return I2C status based on POLL_STATUS_SET_PARAMETERS command.

This is a private method, the **API could change** without previous notice.

Returns

Dictionary with I2C internal details.

```
{
  'rlen' : 65,  <- Value of the requested I2C transfer length
  'txlen': 0,   <- Value of the already transferred (through I2C)␣
↪number of bytes
  'div'  : 118, <- Current I2C communication speed divider value
  'ack'  : 0,   <- If ACK was received from client value is 0, else 1.
  'st'   : 98,  <- Internal state of I2C status machine
  'scl'  : 1,   <- SCL line value as read from the pin
  'sda'  : 0,   <- SDA line value as read from the pin
  'confused': False,   <- see note
  'initialized': True  <- see note
}
```

Hint: If your project does not use I2C, you could reuse SCL and SDA as digital inputs. Call this method to get its logic value.

Note: About **confused** status.

For some reason, ticking SDA line while I2C bus is initialized but idle will cause the next transfer to be bogus. To prevent this, you need to issue a Cancel command before the next *read* or *write* command.

Unfortunately, there is no official way to determine that we are in this situation. The only byte that changes when it happens seems to be byte 18, which is *not documented*.

About **initialized** status:

Same way, calling cancel when the I2C engine has not been used yet will make it to stall and stay in status `0x62` until next reset.

Unfortunately, there is no official way to determine when it is appropriate to call *Cancel* and when it's not. Moreover, MCP2221's I2C status after a reset is different from MCP2221A's (the last one clears the *last transfer length* and the former does not). I found that Cancel fails when byte 21 is `0x00` and works when it is `0x60`. This is, again, *not documented*.

Device.**cmd_retries = 1**

Times to retry an USB command if it fails.

Type
int

Device.**debug_messages = False**

Print debugging messages.

Type
bool

Device.**trace_packets = False**

Print all binary commands and responses.

Type
bool

2.3.11 Exceptions

To capture EasyMCP2221.exceptions you must qualify them as `EasyMCP2221.exceptions`:

```
try:
    mcp.I2C_read(0x51, 1)
except EasyMCP2221.exceptions.NotAckError:
    print("No device")
    exit()
except EasyMCP2221.exceptions.LowSCLError:
    print("SCL low")
```

or import them explicitly:

```
from EasyMCP2221.exceptions import *

...

try:
    mcp.I2C_read(0x51, 1)
except NotAckError:
    print("No device")
    exit()
except LowSCLError:
    print("SCL low")
```

exception NotAckError

I2C slave device did not acknowledge last command or data. Possible causes are incorrect I2C address, device missing or busy.

exception TimeoutError

I2C transaction timed out.

Possible causes:

- I2C bus noise
- incorrect command, protocol or speed
- slave device busy (e.g. EEPROM write cycle)

exception LowSCLError

SCL remains low.

SCL should go up when I2C bus is idle.

Possible causes:

- Missing pull-up resistor or too high.
- Signal integrity issues due to noise.
- A slave device is using clock stretching to indicate it is busy.
- Another device is using the bus.

exception LowSDAError

SDA remains low.

SDA should go up when I2C bus is idle.

Possible causes:

- Missing pull-up resistor or too high.
- Signal integrity issues due to noise.
- An I2C read transfer timed out while slave was sending data, and now the I2C bus is locked-up. Read the Hint.

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.

2.4 I2C Slave helper class

class I2C_Slave(*mcp*, *addr*, *force=False*, *speed=100000*)

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.
- **speed** (*int, optional*) – I2C bus speed. Valid values from 50000 to 400000. See *EasyMCP2221.Device.I2C_speed()*.

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
bytes string

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

bytes string

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...'
```

2.5 SMBus compatible class

2.5.1 Usage

This is a *smbus* compatibility class. You can use it to run any I2C Python library for Raspberry Pi or micropython just using MCP2221's I2C device interface.

Usage:

```
from EasyMCP2221 import SMBus

bus = SMBus(1)
```

or

```
from EasyMCP2221 import smbus

bus = smbus.SMBus(1)
```

2.5.2 Example

In this example, we are using a library from Pimoroni/BME280 to read Temperature, Barometric Pressure and Relative Humidity from a BME280 sensor.

That library is designed for Raspberry Pi or any other system that supports SMBus protocol. It works together with EasyMCP2221 via **SMBus** class.

Install:

```
pip install pimoroni-bme280 EasyMCP2221
```

Example code:

```
import time
from EasyMCP2221 import SMBus
from bme280 import BME280

bus = SMBus(1)
bme280 = BME280(i2c_dev=bus)

while True:
    temperature = bme280.get_temperature()
    pressure = bme280.get_pressure()
    humidity = bme280.get_humidity()
    print('{:05.2f}*C {:05.2f}hPa {:05.2f}%'.format(temperature, pressure, humidity))
    time.sleep(1)
```

Output:

```
17.93*C 933.76hPa 51.57%
17.92*C 933.76hPa 51.57%
17.91*C 933.77hPa 51.53%
17.91*C 933.77hPa 51.50%
17.91*C 933.77hPa 51.54%
...
```

2.5.3 Full reference

Based on kplindegaard/smbus2 interface.

See The SMBus Protocol for more information.

class SMBus(*bus=None*, *force=False*, *VID=1240*, *PID=221*, *devnum=0*)

Initialize and open an i2c bus connection.

Parameters

- **bus** (*int or str*) – (for compatibility only, not used) i2c bus number (e.g. 0 or 1) or an absolute file path (e.g. */dev/i2c-42*). If not given, a subsequent call to `open()` is required.
- **force** (*boolean*) – (for compatibility only, not used) force using the slave address even when driver is already using it.
- **VID** – Vendor Id (default to `0x04D8`)
- **PID** – Product Id (default to `0x00DD`)

block_process_call(*i2c_addr*, *register*, *data*, *force=None*)

Executes a SMBus Block Process Call, sending a variable-size data block and receiving another variable-size response

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Register to read/write to

- **data** (*list*) – List of bytes
- **force** (*Boolean*) –

Returns
List of bytes

Return type
list

close()

(For compatibility only, no effects) Close the i2c connection.

open(*bus*)

(For compatibility only, no effects) Open a given i2c bus.

Parameters
bus (*int or str*) – i2c bus number (e.g. 0 or 1) or an absolute file path (e.g. '/dev/i2c-42').

Raises
TypeError – if type(bus) is not in (int, str)

process_call(*i2c_addr*, *register*, *value*, *force=None*)

Executes a SMBus Process Call, sending a 16-bit value and receiving a 16-bit response

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Register to read/write to
- **value** (*int*) – Word value to transmit
- **force** (*Boolean*) –

Return type
int

read_block_data(*i2c_addr*, *register*, *force=None*)

Read a block of up to 32-bytes from a given register.

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Start register
- **force** (*Boolean*) –

Returns
List of bytes

Return type
list

read_byte(*i2c_addr*, *force=None*)

Read a single byte from a device.

Return type
int

Parameters

- **i2c_addr** (*int*) – i2c address
- **force** (*Boolean*) –

Returns
Read byte value

`read_byte_data`(*i2c_addr*, *register*, *force=None*)

Read a single byte from a designated register.

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Register to read
- **force** (*Boolean*) –

Returns
Read byte value

Return type
int

`read_i2c_block_data`(*i2c_addr*, *register*, *length*, *force=None*)

Read a block of byte data from a given register.

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Start register
- **length** (*int*) – Desired block length
- **force** (*Boolean*) –

Returns
List of bytes

Return type
list

`read_word_data`(*i2c_addr*, *register*, *force=None*)

Read a single word (2 bytes) from a given register.

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Register to read
- **force** (*Boolean*) –

Returns
2-byte word

Return type
int

`write_block_data`(*i2c_addr*, *register*, *data*, *force=None*)

Write a block of byte data to a given register.

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Start register
- **data** (*list*) – List of bytes

- **force** (*Boolean*) –

Return type
None

write_byte(*i2c_addr*, *value*, *force=None*)

Write a single byte to a device.

Parameters

- **i2c_addr** (*int*) – i2c address
- **value** (*int*) – value to write
- **force** (*Boolean*) –

write_byte_data(*i2c_addr*, *register*, *value*, *force=None*)

Write a byte to a given register.

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Register to write to
- **value** (*int*) – Byte value to transmit
- **force** (*Boolean*) –

Return type
None

write_i2c_block_data(*i2c_addr*, *register*, *data*, *force=None*)

Write a block of byte data to a given register.

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Start register
- **data** (*list*) – List of bytes
- **force** (*Boolean*) –

Return type
None

write_word_data(*i2c_addr*, *register*, *value*, *force=None*)

Write a single word (2 bytes) to a given register.

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Register to write to
- **value** (*int*) – Word value to transmit
- **force** (*Boolean*) –

Return type
None

2.6 Limitations and bugs

2.6.1 Chip or software limitations

USB speed limits

MCP2221's command rate is limited by USB polling rate. Each command requires two USB slots. One to send the command to the device, and the other to get the response. See *Internal details* for details.

On xHCI controllers (USB 3.0), polling rate is 1000Hz. So you can issue one command every 2ms. That command could be a DAC update, ADC read, GPIO, I2C operation, etc.

- GPIO update rate `GPIO_write()`: 500Hz
- GPIO read rate for `GPIO_read()`: 500Hz
- ADC sample rate for `ADC_read()`: 500Hz.
- DAC update rate for `DAC_write()`: 500Hz.

On eHCI (USB 2.0), the maximum update rate I measured is 333 commands per second.

These ratios depend on multiple parameters. Like your USB hardware (including cable and hub), operating system, or the number of devices connected to the same bus.

I2C speed limit

Each I2C interaction requires multiple USB commands. See *Internal details* for details.

Sending one byte will require: setup, send data, and get the result. Reading one byte will require: setup, finish test, and read data.

Depending on your USB polling rate, each of these commands can take 2ms or more.

I2C speed (100kHz / 400kHz) only matters when you are transmitting a lot of bytes in a row. For a few bytes interaction, speed is limited by the USB polling rate.

Internal reference reset

From MCP2221A's datasheet (section 1.8):

When the Set SRAM settings command is used for GPIO control, the reference voltage for VRM is always reinitialized to the default value (VDD) if it is not explicitly set.

This is compensated by software. But, due to the calling interval, there will be always a 2ms gap in the DAC output if it is using internal reference (not Vdd) when you change any pin function.

I2C crashes

Eventually, due to a glitch or unexpected timeout, the MCP2221 cancels an I2C transfer. The slave may be in the middle of sending a byte, and expecting some clocks cycles to send the rest of the byte.

MCP2221 is unable to start a new I2C transfer while SDA line is still busy. And the slave won't release SDA until next clock cycle. So the whole bus hangs.

See *LowSDAError*.

Misc

- 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.
- This library does not work with password protected devices. You cannot use it to set or clear MCP2221's Flash password.
- Changing VID/PID not supported.
- Maximum length for single I2C read or write operations is 65535 bytes.
- Regardless of the output frequency, MCP2221(A)'s clock output has glitch every 1ms.

2.6.2 Software Bugs

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

2.7 Internal details

USB transactions are always initiated by the Host. MCP2221 supports Full-Speed USB at 1000Hz polling rate. That is, a transaction every one millisecond.

Each USB transaction can be:

Output:
: the host will send data to the peripheral. Used for commands.

Input:
: the host wants the peripheral to send data. Used for replies.

We need 2 USB transactions (2ms) for each command/reply. Thus, the highest DAC update rate for or ADC sampling rate is 500Hz.

The maximum length of a data payload is 64 bytes.

2.7.1 I2C transfers

Write transfer

Fig. 1: Timeline of a 100 bytes I2C **write**. Open the image in a new tab to see it full size.

While the I2C engine is sending `data1`, subsequent write commands will fail and will be ignored. Only when `data1` has already been sent, the device responds with OK and proceed to send the next data chunk `data2`.

Read transfer

Fig. 2: Timeline of a 75 bytes I2C **read**. Open the image in a new tab to see it full size.

While the MCP2221 is reading bytes, subsequent calls to *Read I2C data buffer* (40) will fail. When the buffer is ready (or full), the call succeed, the data is returned and the reading of next chunk begins.

Transfer failure

Note that since the USB host only send or requests data at fixed intervals of 1 ms, the state of the last issued command may or may not match the actual state of the I2C transfer.

For example, in a read operation to a nonexistent device. The I2C transfer starts and even ends before the response to the write command `91` is read.

Fig. 3: I2C read failure. Not acknowledge.

Moreover, like in the above successful reading, the *I2C read command* (91) succeed but the *I2C read data command* (40) fails, exactly as before.

I use internal I2C engine status code to differentiate between both cases. Unfortunately, not all the states are fully documented.

2.8 Changelog

2.8.1 V1.6

1.6.3

I2C:

- New *SMBus compatible class*. Useful to use other Python I2C modules with MCP2221 interface.
- Fixed. I2C slave class exception when device is not present.

Documentation:

- Conflict with kernel module `hid_mcp2221`. See *Delay at the end of script (Linux)* in *Install / troubleshooting*.
- Explain I2C speed limit for very short transfers.

- Document *SMBus compatible class*. Include example code for BME280 (Temperature, Pressure, Humidity sensor).

V1.6.2

ADC/DAC:

- Fixed bug: when ADC reference is VDD and DAC reference is VRM and a new GPIO configuration is applied, DAC stops working. It seems to be related with a known MCP limitation.
- Fixed bug: Restore DAC reference value after reset.
- In some cases MCP2221A's firmware does not restore DAC or ADC pin assignment when it boots. Software workaround.

I2C:

- Deprecated `I2C_cancel()` and `I2C_is_idle()`. Bus is now managed automatically. You can use `_i2c_release` and `_i2c_status` as replacement if needed.
- Fixed. Low SCL and low SDA exceptions were swapped.
- When the I2C bus detects SDA activity, the next transfer does not work fine. Prevented via software. See `_i2c_status`.

Documentation:

- Instructions and schematic for testing in the *Install / troubleshooting* section.
- Replaced PNG schematics by SVG versions in *Examples*.
- Troubleshooting section to run as an unprivileged user in Linux (udev rule).
- Added developers section with details about I2C transfers. See *Internal details*.

Misc:

- Added test suite.
- Added IOC edge detection setting in device representation.
- Fixed. Bug when reset a device with customized VID/PID.
- Multiple tries to find the device after a `reset()` (until timeout).

V1.6.1

Improved USB stability:

- Added timeout in HID read.
- Added retries in send_cmd.
- Better USB trace_commands output format.
- Removed sleep parameter in `send_cmd()`.

GPIO / ADC and DAC:

- GPIO output values given with `GPIO_write()` function are now preserved when calling `SRAM_config()` (like to change DAC value, or pin function).
- Restore ADC/DAC Vref when calling `SRAM_config()` (see *Limitations and bugs*).
- Reload Vrm after power-up according to saved configuration.

More reliable I2C functions:

- Rewritten *I2C_read()* to take into account internal I2C engine status.
- Rewritten *I2C_write()* to prevent infinite loop, quicker write and ACK checking.
- Timeout and early failure check in read and write to prevent infinite loop.
- Custom exceptions for better error handling (see *Exceptions* in *Full API reference*)
- Automatically try to recover from an I2C error in past operation.

New features:

- Function to save current state: *save_config()*.
- Added speed parameter in I2C Slave class.

Documentation:

- Removed *self* argument from *autodoc* methods.
- Added pictures and schematics.
- Added MCP2221 pinout guide.
- Added advanced ADC/DAC examples section.
- Added license.
- Corrected typos.
- Formatting.

V1.6.0

Released 1.5.1 again by mistake.

2.8.2 V1.5

V1.5.1

Add I2C Slave helper class.

V1.5.0

First EasyMCP2221 version.

2.8.3 Older releases

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

I made 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.

2.9 Links

Source GitHub repository:
https://github.com/electronicayciencia/EasyMCP2221

MCP2221 and MCP2221A - USB 2.0 to I2C/UART Protocol Converter with GPIO - Microchip Inc.
https://www.microchip.com/en-us/product/MCP2221

https://www.microchip.com/en-us/product/MCP2221A

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

Symbols

_i2c_release() (*in module EasyMCP2221.Device*), 38
_i2c_status() (*in module EasyMCP2221.Device*), 39

A

ADC_config() (*in module EasyMCP2221.Device*), 28
ADC_read() (*in module EasyMCP2221.Device*), 27

B

block_process_call() (*SMBus method*), 45

C

clock_config() (*in module EasyMCP2221.Device*), 35
close() (*SMBus method*), 46
cmd_retries (*Device attribute*), 39

D

DAC_config() (*in module EasyMCP2221.Device*), 29
DAC_write() (*in module EasyMCP2221.Device*), 28
debug_messages (*Device attribute*), 39
Device (*class in EasyMCP2221*), 23

E

enable_power_management() (*in module EasyMCP2221.Device*), 36

G

GPIO_read() (*in module EasyMCP2221.Device*), 26
GPIO_write() (*in module EasyMCP2221.Device*), 26

I

I2C_cancel() (*in module EasyMCP2221.Device*), 33
I2C_is_idle() (*in module EasyMCP2221.Device*), 34
I2C_read() (*in module EasyMCP2221.Device*), 31
I2C_Slave (*class in EasyMCP2221.I2C_Slave*), 41
I2C_Slave() (*in module EasyMCP2221.Device*), 30
I2C_speed() (*in module EasyMCP2221.Device*), 32
I2C_write() (*in module EasyMCP2221.Device*), 30
is_present() (*I2C_Slave method*), 42

L

LowSCLError, 40
LowSDAError, 41

N

NotAckError, 40

O

open() (*SMBus method*), 46

P

process_call() (*SMBus method*), 46

R

read() (*I2C_Slave method*), 42
read_block_data() (*SMBus method*), 46
read_byte() (*SMBus method*), 46
read_byte_data() (*SMBus method*), 47
read_i2c_block_data() (*SMBus method*), 47
read_register() (*I2C_Slave method*), 42
read_word_data() (*SMBus method*), 47
reset() (*in module EasyMCP2221.Device*), 37

S

save_config() (*in module EasyMCP2221.Device*), 25
send_cmd() (*in module EasyMCP2221.Device*), 38
set_pin_function() (*in module EasyMCP2221.Device*), 24
SMBus (*class in EasyMCP2221*), 45
SRAM_config() (*in module EasyMCP2221.Device*), 37

T

TimeoutError, 40
trace_packets (*Device attribute*), 40

W

wake_up_config() (*in module EasyMCP2221.Device*), 36
write() (*I2C_Slave method*), 43
write_block_data() (*SMBus method*), 47
write_byte() (*SMBus method*), 48

write_byte_data() (*SMBus method*), 48
write_i2c_block_data() (*SMBus method*), 48
write_register() (*I2C_Slave method*), 43
write_word_data() (*SMBus method*), 48