

# PYORCAI2C

---

## INSTALL

```
pip install pyorcai2c
```

## BASIC USAGE

```
import pyorcai2c
i2c = ftdi(b'SERIAL_NUMBER')

res = i2c.write(slave=ANY_NUMBER, target=ANY_NUMBER, data=ANY_NUMBER)
print(res)

res = i2c.read(slave=ANY_NUMBER, target=ANY_NUMBER)
print(res)
```

## RETRIEVING FTDI BOARD SERIAL NUMBER

If you don't know the serial number of the FTDI board you are using do the following:

```
import ftd2xx
available_devices = ftd2xx.createDeviceInfoList()
available_devices = ftd2xx.listDevices()
```

## BURST READ WRITE USAGE

`i2c.write` always requires the `slave` argument as a integer positive number. No other arguments are mandatory.

```
# ping slave
res = i2c.write(slave=ANY_NUMBER)
print(res)

# i2c command
res = i2c.write(slave=ANY_NUMBER, target=COMMAND_AS_A NUMBER)
print(res)
```

`target` argument of `i2c.write` can either be a single integer or a dictionary in the form `address_as_integer: data_as_integer`. `i2c.write` will optimize i2c communication by performing

the minimum number of burst write needed to complete the request.

If `target` argument of `i2c.write` is a number `data` argument is also needed. If `target` argument of `i2c.write` is an `{int: int}` dictionary the `data` argument will be ignored.

A burst read of n bytes can easily be performed through one of the following ways:

```
# read burts by specifing start address and number of bytes
res = i2c.read(slave=ANY_NUMBER, target=ANY_NUMBER, n=n)
print(res)

# read burst by specifing target as an array
res = i2c.read(slave=ANY_NUMBER, target=[ANY_NUMBER_1, ANY_NUMBER_2,
ANY_NUMBER_3])
print(res)
```

`target` argument of `i2c.read` can either be a single integer or a list of integers. `i2c.read` will optimize i2c communication by performing the minimum number of burst read needed to complete the request. If `target` argument of `i2c.read` is a number `n` argument is defaulted to 1 if absent. If `target` argument of `i2c.read` is a list of integers the `n` argument will be ignored.

## I2C USAGE BASED ON REGISTER MAP

To import ORCA products register map do the following:

```
import os
cwd = os.getcwd()
regmap_filepath = os.path.join(cwd, 'regmaps', 'pmic01.json')
regmap = i2c.load_register_map(regmap_filepath)
```

Loading a register map will allow for both register based and field based communication.

```
res = i2c.write(slave=ANY_NUMBER, target=VALID_REGISTER_NAME,
data=ANY_NUMBER)
print(res)

res = i2c.write(slave=ANY_NUMBER, target=VALID_FIELD_NAME, data=ANY_NUMBER)
print(res)

res = i2c.read(slave=ANY_NUMBER, target=VALID_REGISTER_NAME)
print(res)

res = i2c.read(slave=ANY_NUMBER, target=VALID_FIELD_NAME)
print(res)
```

## ADVANCED USAGE:

All of the above ways of communications can be mixed up without much restraint and in a pretty natural way. Mixing up `integers`, `register_names` or `field_names` as the target arguments.

The following strange calls will simply and just work. The module will take care of optimizing the I2C communications by minimizing it through as many burst command as possible.

The module response structure will reflect the request for the data part of the response while the acks part will be based on starting register address of the burst command.

```
res = i2c.read(
    slave=slave,
    target=[
        0x04,
        0x02,
        0x05,
        0xA1,
        0xA3,
        0xA2,
        0x06,
        'ChargeCtrl1',
        'ChargeCtrl2',
        'LD01Mode',
        'LD02Ctrl',
        'LD02Voltage',
        'tst_bias_a',
        'TstCntrl7',
        'LD01Voltage'
    ]
)
print(res)

res = i2c.write(
    slave=slave,
    target={
        0x02:0xAA,
        0x01:0xEA,
        0x10:0xCD,
        0x05:0xFE,
        0x06:0xBB,
        'LD02Voltage':0x55,
        'Buck1Ctrl2':0x33,
        'unused_Buck1Ctrl2_b2':1,
        'unused_Buck1Ctrl2_b4':1,
        'unused_Buck1Ctrl2_b5':0,
        'Buck1Ton':255,
        'Buck1VRegA':0
    }
)
print(res)
```

## additional code examples:

You can find additional examples

here: <https://github.com/orcasemi/pyorcai2c/blob/main/tests/test.py>

and here: <https://github.com/orcasemi/pyorcai2c/blob/main/tests/debug.py>

## co-development (ORCA DEVELOPMENT TEAM ONLY)

1. clone the repository locally

```
git clone git@github.com:orcasemi/pyorcai2c.git
```

if you get an error about not having enough clearance to clone it please follow [this tutorial](#) to setup a github ssh-key and contact orcasemi github organization to be added as a member

2. setup and activate virtual environment

```
cd pyorcai2c  
python -m venv
```

3. if you are on windows and you have never done that before you need to enable powershell to run scripts in order to activate the newly created virtual environment

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

run that from an admin elevated powershell

4. activate the virtual environment

```
.\venv\Scripts\activate
```