Remoteio

Content table

1. Introduction

The idea to manage gpiozero devices from a remote computer is not new. On Raspbery Pi this is realized by remote gpio, but not for Raspberry Pi Version 5. In order to fill this gap there are two possibilities:

- modification of remote gpio
- establishing a new feature

The first one would be the simplest one, because in the new Raspbian OS there is a link from /dev/gpiochip4 to /dev/gpiochip0 solving most problems of gpio access that arose with the introduction of Raspberry Pi 5.

Because no attempt is made to adapt remote gpio to this new situation a new feature called remoteio is created. This new feature has two main ideas:

- delegating tasks to gpiozero on the remote server
- extension to non gpiozero devices
- extensibility to further devices

Remoteio is realized with Python language.

The idea of delegating to gpiozero is very practical because the device objects use exactly the parameters of gpiozero as described in the API of gpiozero.

The idea of an extension to non gpiozero devices needs an adaption or wrapping of existing device approaches to remoteio. Especially a value-property is needed. So when an existing device access does not have a value-property but a getValue-function we need a wrapper-class, with the original class as superclass, that uses getValue of the super class to define the value-property. There are many examples given, how to proceed.

Because user want to use remoteio for their own device classes remoteio uses the folders remoteio_wrapper and remoteio_extensions. Here the user can add his own devices or they may make them available for publishing in remoteio.

1.1. Overview of supported Devices

At the state of the art not all gpiozero devices are supported. There are also some non gpiozero devices supported.

Gpiozero devices:
AngularServo
Button
DistanceSensor
LED
LEDBarGraph
LEDBoard
LightSensor
LineSensor
MCP3208
MotionSensor
Motor
PWMLED
PhaseEnableMotor
RGBLED
Servo
TonalBuzzer

non gpio zero devices:
ADS1115
MCP23017,MCP23S17,MCPLED,MCPButton
MCP4801,MCP4802,MCP4811,MCP4812,MCP4821,MCP4822,MCP4902,MCP4912,MCP4922
W1ThermDevice

only on client side (compositum design pattern):
Remote_LEDCompositum
Remote_Kontext

Detailed examples are found in controller.py or at the end of the corresponding .py-files

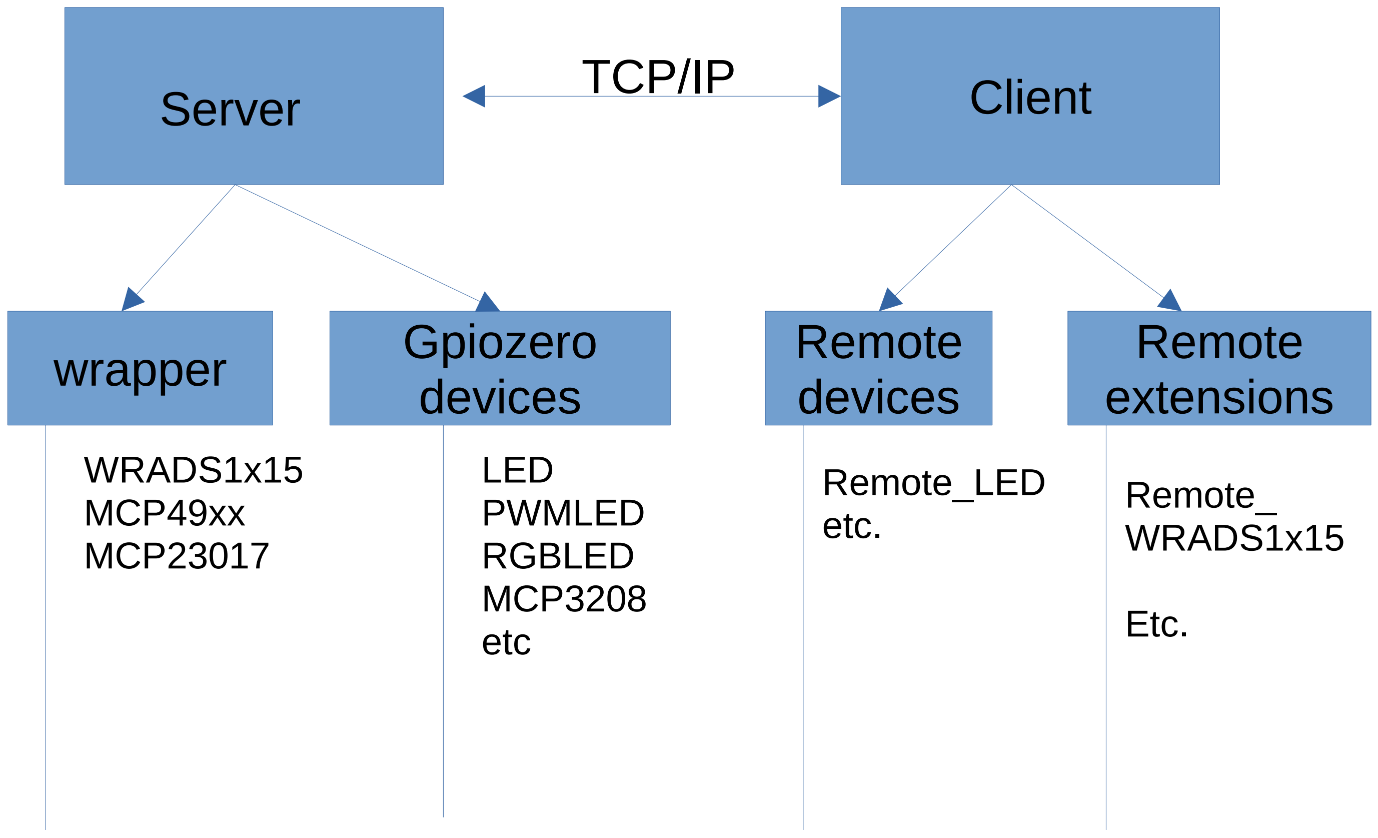
2. Base concept

Remoteio has two kernel modules that should not be modyfied by the user:

- remoteio_server.py
- remote_client.py

remoteio_server is implemented on the server computer and is written in a universal way treating all devices in the same manner. It needs no special knowledge of the devices. For each device a process without knowledge of the device is used as proxy to manage the device.

Server and Client are connected by a TCP/IP connection (default port 8509).



3. Programming

The main idea is delegating of tasks to devices on the server.

To
def __init__(*args,**kwargs) of a device on the server side
corresponds
def __init__(remote_server, ident, obj_type, *args,**kwargs) on the client side.
A method like on() or off() has the same parameter on the server side and the client side.

For the __init__ function of a remote device on the client side we need the object remote_server for the TCP/IP connection, further an ident for identification and an obj_type in order to create the right device on the server. The user needs only to generate the remote_server object by

    rs=remote_server(ip_address,port)

and then e.g.

    rled=Remote_LED(rs,*args,**kwargs)

Ident and obj_type are added by the superclass Remote_DigitalDevice in remoteio_client.py during the creation of the remote device in concern. Idents are administrated by the class Remote_Supervisor in remote_client.py.

The use of generators and the decorator property make it possible to have a gpiozero-like user interface. Properties like source, when_pressed or functions like wait_for_release are also realized for the remote devices (in an abstract way by the classes Remote_DigitalDevice and its superclass Remote_Device, both in remote_client.py)

Extensions to non-gpiozero devices are treated in the same manner because these extensions are also treated as Remote_DigitalDevices.

It is important to understand how parameters are interpreted.

Our goal is to transfer expression like **"param=python-expression"** to the server side, where it is needed. This expression is well interpreted on server side by the Python-Interpreter.

We consider param=python-expression on client side. This expression is normally used only on client side and the Python-interpreter interprets the text python-expression in the .py file.

Remoteio sends **"param="+ str(python-expression)** to the server side.

When the python-expression is not overinterpreted by the Python-interpreter, i.e. if

> **str(python-expression)=='python-expression'** or
> **str(python-expression)=="python-expression"**

then remoteio sends "param=python-expression" that is well interpreted on server side.

This is the case where the python-expression is a number, False or True or a tuple,list or dict consisting only of numbers ,True or False or strings, the usual case for gpiozero devices and extensions. You may set the parameters as for working only on the client side.
If

str(python-expression)!='python-expression'

as rule of thumb you write

param="python_expression" (interpretation on server side)
param="\"" + python_expression + "\ "" (interpretation on client side)

in quotations or quotations as escape sequences.

Examples:

For testing copy the following examples in the shell IDLE(Python).

1. simple examples

param=1.0

str(1.0)=='1.0', (by Python 1.0 is interpreted as float) is ok

param=[1.0,'hello']

str([1.0,'hello'])=="[1.0,'hello']" is ok

param=[1.0,”hello”]

str([1.0,"hello"])=="[1.0, 'hello']" is ok

if x=='hello' and param=[1.0,x]

str([1.0,x])== "[1.0, 'hello']" is ok

2. param=x, where x='hello' ?

str(x)='hello' != 'x'

interpretation on client side needed, you write

param="\"" + x + "\""

str("\"" + x + "\"")=='"hello"' with double quotations inside, is ok

3. param='hello' ?

"param="+ str(python-expression) ="param=hello" (tested with Idle(Python))

On server side hello is interpreted as a Python object, that in best case does not exist on server side.

str('hello') ='hello' != ''hello'' (two single quotations) and

str('hello') ='hello' != "'hello'" (double and single quotations) interpretation only on server side needed:

param= "'hello'" (double quotations outside, single quotations inside)

str("'hello'")="'hello'" (double quotations outside, single quotations inside)

and

"param="+ str(python-expression)=="param= 'hello'" is ok

A string as parameter value must be set in double quotations

4. param=Color('red') ?

The Python-interpreter resolves str(Color('red')) on the client side. After ‘from colorzero import Color' we have str(Color('red'))=='#ff0000' and "param=#ff0000" that cannot be interpreted on server side. In this case, the resolution of the Python-expression on client side must be suppressed.

Therefore **param="Color('red')"** and only the server side interprets the expression.

5. What to do with param=(1.0,x,Color('red')), where x='hallo'?

Here we have no general solution, because x must be resolved on client side but Color('red') isn't allowed to be resolved on client side. You must compose the string.

y=str((1.0,x))[0:-1]+","+"Color('red')"+")"

and then param="\""+y+"\""

str("\""+y+"\"")**==** '"(1.0, \'hello\',Color(\'red\'))"' (single quotations outside)

"param="+ str(python-expression)=='param="(1.0, \'hello\',Color(\'red\'))"' is ok

Fortunately the user interface doesn't have this problem.

Considering gpiozero devices we must know the above handling for Tonalbuzzer and RGBLED.

Considering wrapper-devices we must know the above handling for W1ThermSensor. Note that the solution is given in the examples.

Using the examples given in remoteio there will be no problem of application.

There is a module remoteio_helper.py where many useful functions are established and used by remoteio.

remoteio_server uses setattr, getattr and eval to abstract from special devices. This is accomplished by naming special classes, constants needed in remoteio_server, in the __init__.py file of the folder remoteio_wrapper that only works on server side.

The package remoteio has subpackages:

The folder remoteio contains:

folders: remoteio_devices, remoteio_extension, remoteio_wrapper each folder has its own __init__.py file.

moduls (.py files): __init__.py, remoteio_constants.py, remoteio_helpers.py,

remoteio_client.py, remoteio_server.py

The names that you want load directly to remoteio are those in the __init__.py files.

__init__.py of folder remoteio:

```
#!/usr/bin/env python3
from .remoteio_constants import *
from .remoteio_helper import *
from .remoteio_client import RemoteServer,RemoteDigitalDevice,RemoteSupervisor
from .remoteio_server import run_server
from .remoteio_devices import *
from .remoteio_extensions import *
from .remoteio_wrapper import *

import logging
logger=logging.getLogger(__name__)
```

Best is explicitely naming what you want to have in the namespace.

The constants we want to have all. The helper functions have the necessary imports in the function bodys, so that there are only the names of the functions imported.

The __init__.py files of the folders remoteio_devices, _extensions, _wrapper give the exact information what you can get from the modules in concern. Naturally you can acces to any class, constant, function or method by fully qualifying the access, but ‘from remoteio import *' only furnishes what is allowed.

A simple program:

```
from remoteio import *

print(dir())
```

shows all possible names in the namespace of remoteio.

In the modules of the remoteio package all accesses to remoteio-classes are fully qualified. The reason is that you must pay too much attention on import failures otherwise (e.g. remoteio namespace is sometimes only partially loaded or you have a circular importing).

The logging is defined in the above way, because you must understand logging as a tree of loggers where each logger gives a local information on the place where the information is given. The exact setting of the logger is given in the main program. Each module begins with the same logging script as that in the __init__.py file. By this way it is possible to exactly find the place where the logger has given an information. Especially the line number is given back if desired in the main program. This facilitates the finding of programming errors.

When making a second TCP/IP connection with the same port to the server, then the second connection interrupts the first one on server side.

4. remoteio_wrapper.py

Existing device classes that are not supported by gpiozero perhaps have to be adapted to work with remoteio. At this purpose the design pattern of adapter or wrapper is useful. The value property is needed for remoteio:

```
WRDEVICE=class(DEVICE)
        ….

@property
def value(self)
        return self.getValue()
```

when getValue() is the function of DEVICE that furnishes the value.

You can then get the value by

x=WRDEVICE.value on the server side or

x=Remote_WRDEVICE.value on the client side

To each family of devices realized by the same superclass there exists a Python-Module (.py-file).

E.g. in the directory .../remoteio/remoteio_wrapper there exists mcp49xx.py containing corresponding classes like MCP4801 or MCP4922. The module mcp49xx.py must be registerd in the __init__.py of remoteio_wrapper as well as the classes that you want to import from there.

By creating a suitable device module and registering in __init__.py of remoteio_wrapper folder the user may adapt an own device for the use by remoteio. There are many examples realized.

4.1. W1ThermDevice

class W1ThermDevice(W1ThermSensor):
def __init__ (self,*args,**kwargs) with the parameters of the superclass

from w1thermsensor import W1ThermSensor,Sensor,Unit

sensor=W1ThermDevice(sensor_type=Sensor.DS18B20, sensor_id='00000cb6ad51')

has properties:

- value
- resolution
- available sensors

This class makes it possible to observe values of a W1ThermSensor device by remoteio.

4.2. MCP49xx

DAC: Digital-Analog-Converter, may be used as digital Potentiometer

SPI-device

Modul mcp49xx.py in the directory remoteio_wrapper

MCP49xx is superclass of MCP4801, MCP4811, MCP4821, MCP4802, MCP4812, MCP4822, MCP4902, MCP4912, MCP4922 which are all defined in mcp49xx.py

MCP48?1(bus:int=0,device:int=0)

MCP48?2(channel:int,bus:int=0,device:int=0)

MCP49?2(channel:int,bus:int=0,device:int=0)

properties:

- value
- resolution
- gain
- buf (only with effect on MCP49xx)

Remoteio interpretes the name of the class to get the resolution or number of channels. The answers given may be wrong, if you use the false class for a chip. The chip only behaves as described in the datasheet.

For devices with only one output channel, channel is not a parameter.

For devices with more output channels it is obligated.

The devices of the 48xx-serie have an internal Vref and are not dependent of fluctuations of the electric tension furnished by the Raspberry Pi. For the 49xx series remoteio reduces the maximum input value to 2**resolution – 2 instead of 2**resolution – 1. The reason is that fluctuations in Vref or VDD make a jump from maximum to zero (or blinking) impossible. With 4094 as maximum instead of 4095 in some attempts this was each time possible. There were no problems with the 48xx series.

Studying the source code it should seem strange that we have an own access to SPI for each channel with the same device number. Note that remoteio writes data in only one register of the device, a non interruptable action.

The access to the same SPI-device by different threads must be secured by a Lock to avoid conflicts. It seems that SPI has here its own mechanism and perhaps such a Lock is not necessary.

The problem of conflicts is the same whether using one or two SPI-Accesses for the same device (here for different channels). The idea to this proceeding is taken from gpiozero where the same is made for MCP3208. Remoteio uses it also for the ADS1x15-series (I2c-Bus). But the chips MCP23S17 (SPI) and MCP23017 (I2C) use one register where each bit represents a pin. Here an internal Lock by remoteio is necessary, but SPI or I2C conflicts not provoked by remoteio cannot be solved by remoteio.

Because a bit has only two states MCP23x17-pins cannot pulse.

Example:

m=MCP4922(0,device=1)

m1=mcp4922(1,device=1)

m.close()

m.input=1000 # raises an error

m1.input=1000 # no error

These devices can be used like digital potentiometers

4.3. WRADS1x15

Analog-Digital_Converter (ADC)

I2C device (default address 0x48)

Modul wrads1x15.py in the directory remoteio_wrapper

The Adafruit library is used.

wads=WRADS115(self,*channelNr,**kwargs)

one channel for single mode, two channels for differential mode

The corresponding mode is automatically recognized by the underlying Adafruit components

by the number of channels.

parameter by name:

*channelNr, gain, data_rate, mode,address

Best is to use *channels as positional parameters and the others by name.

```
  ads0=WRADS1115(0,1,address=int(0x48))

  ads1=WRADS1115(1,3,address=int(0x48))
  print(hex(ads0.address))
  print(ads0.value)
```

For each channel there is an i2c-access (naturally to the same i2c-address).

```
def __init__(…):
    …
    sleep(0.001)
```

You need a wait of 1 ms in order to initialize the chip. Otherwise the first value may be wrong.

The ADS1115 is useful, because the Raspberry Pi has no analog input devices

4.4. MCP23x17

Modul mcp23x17.py in the directory remoteio_wrapper

There exists a SPI-version of the chip, named MCP23S17 and a I2C-version named MCP23017.

Both chips furnish 16 further digital pins for the Raspberry Pi. Both chips are supported by remoteio. For the use by remoteio corresponding classes are created, where the pins have on, off, blink functions.

Further properties like source, when_pressed,when_released or methods like wait_for_press, wait_for_release are also supported. Buttons may be created with a bounce-time.

Contrarily to the approaches before, pins are not looked upon as objects that directly work with SPI or I2C. The communication with the bus system is performed by the classes MCP23017 resp. MCP23S17 and the pins depend on these classes. Note that a change of only one bit (state of a pin) of the GPIO-register needs a read of the register, a change of the bit and then a writing of the register. Such an action is interruptable. After a read of the register another one can change the register before you realize your change of data by a write. And when you write in the register the changes of the other one are overwritten. Therefore a change of the register must be secured by a lock.

The IOCON-register is an eight bit register that can be addressed by the two addresses 0x0A and 0x0B (default settings after start or restart of the chip). It is somehow confusing that a register may be addressed by different addresses. Remoteio uses ICON.BANK=0 (default) and ICON address 0x0A.

Creation:

mcp=MCP23017(busId:int=1,address:int=0x20)

address=I2C-Address (0x20 – 0x27, depending on hardware address pins) mcp=MCP23S17(port=0, device=0, address=0x00):

address = 0x00 – 0x07 depending on hardware address pins

In the case of MCP23S17 address pins may be disabled by setting bit 3 of the register IOCON to 0.

Address pins are enabled by remoteio . So you can use eight MCP23S17 on the same port and device.

In remoteio this is realized by: mcp.setRegisterBit(IOCON,3,1).

It is recommended to consult the datasheet of MCP23x17 or the Raspberry Pi book of Kofler, Kühnast and Scherbeck in Rheinwerk-Verlag for a summary.

MCPLED and MCPButton are created as follows:

mled=MCPLED(mcp:MCP23x17,pinNr:int)

mb   = MCPButton(mcp:MCP23x17,pinNr:int,bounce_time:int=None)

It takes no matter, whether mcp is an object of MCP23S17 or MCP23017.

The reason is the following:

All actions are made by the superclass MCP23x17. The differences between SPI and I2C are treated by Mixin-classes and multiple inheritance:

MCP23017(MCP23x17,Mixin23017) and MCP23S17(MCP23x17,Mixin23S17)

MCP23x17:

methods: reset(),
setValue(pinNr,val),
on(pinNr,**kwargs),
off(pinNr,**kwargs),
toggle(pinNr,**kwargs),
blink(pinNr,**kwargs),
create_MCPLED(pinNr),
create_MCPButton(pinNr,bounce_time:int=None),
set_hold_time(pinNr:int,hold_time:int),
set_hold_repeat(pinNr:int,hold_repeat:bool)

properties: is_held,held_time,hold_time,hold_repeat
iodir, iopol, gpinten, defval, intcon, iocon, gppu, intf, intcap, gpio, olat,
(access to the corresponding registers A,B as 16-bit word)

value, values

MCP23017 and MCP23S17:

methods: close()

MCPLED: off(),

on(on_time=None)

toggle()

blink(on_time=1,off_time=1,n=None,background=True)

_source_delay=0.005 (default for values generator)
for finer sampling rates you can choose shorter delays
gpiozero works with 0.002 sec

Note that blink is supported by a process that must be terminated at the end of the program.

server_start.py uses the necessary techniques. When the TCP/IP connection is interrupted remoteio makes automatically the necessary closing.

MCPButton:

methods: wait_for_release(timeout:int=None)

wait_for_press(timeout:int=None)

properties:

when_pressed, when_released, when_held,
value, values,
is_active, is_pressed, is_released, is_held,
held_time, hold_time, hold_repeat

gppu, iodir, ipol used on pin-level i.e. values are 0,1

sampling rates for generators :
_source_delay=0.01 (default for values generator)

gen_when_delay=0.01 (default for gen_when generator
gen_wait_delay=0.01 (default for gen_wait generator)
for finer sampling rates you can choose shorter delays
gpiozero works with 0.002 sec

The MCP23x17-Chips are by construction more for use by a state-machine. In this context you may use data=mcp.gpio to get the values of all 16 pins as bits of data. Then you change data and realize the change by mcp.gpio=data. The approach by remoteio supports further changing only one pin in order to have the best approach to gpiozero. The event-handling of gpiozero is imitated.

Buttons are better supported. Note that the values of buttons cannot be changed. Using a state machine, you must put the buttons on an own MCP23x17 chip for work on register level.

5. remoteio_extensions.py

In this chapter the remote acces to (non gpiozero) wrapper-devices is explained.

5.1 Remote_W1ThermDevice

sensor=Remote_W1ThermDevice(remote_server:RemoteServer,*args,**kwargs)

e.g.
sensor=Remote_W1ThermDevice(rs,sensor_type='Sensor.DS18B20', sensor_id="'00000cb6ad51'")
For the usage of quotations consult chapter 3.
*args, **kwargs are arguments of W1ThermDevice on the server side in the sense of delegating tasks to the original device.

Properties: value:float, values:generator, resolution:int, available_sensors:list

temperature = sensor.value

A signalization for too hot temperature may be realized as follows:

```
RLED=Remote_LED(rs,21)
tem_gen_delay=0.01
def temp_gen():
    while True:
        try:
            sleep(tem_gen_delay)
            if sensor.value>30
                yield 1
            else:
                yield 0
        except:
            break

RLED.source=temp_gen()
```

5.2. Remote_MCP49xx

These devices do not send anything to the program. You only put input values there. For a test whether the select pin etc. is well defined, you may use a test program like the following:

```
#Connect a led with an output pin of MCP4811 (MCP4812)

rmcp=Remote_MCP4811(rs,device=1,max_speed_hz=250000)
#rmcp=Remote_MCP4812(rs,0,device=1,max_speed_hz=250000) # channel needed
while True:
        for i in range(1024):
                rmcp.value(i)
        for i in range(1023,-1,-1):
                rmcp.value(i)
```

In the 48xx series with an internal reference of 2.048V the output line can offer 4.096V (by gain=2),

more than the 3.3V of the Raspberry PI. You can use these devices as potentiometer.

The DAC output amplifier of each channel of MCP49x2 can drive the output pin with a range of VSS to VDD. So with VDD=3.3V of Raspberry PI you cannot destroy an input pin.

Properties: value(read/write) , values (generator), gain(read/write), resolution(read only)

5.3. Remote_WRADS1115

Modul remote_wrads1x15.py in folder remoteio_wrapper.

properties: value , values, address

methods: gen_voltage() → generator-function

ADS115 is a 16-bit analog-digital Converter (ADC)

The ADS111x have an integrated voltage reference. Analog input channels should have a voltage <=VDD.

In the case of Raspberry Pi with VDD = 3.3V and full-scale range = ±4.096V, a full-scale ADC output code cannot be obtained. For example, with VDD = 3.3V and FSR = ±4.096V, only differential signals up to VIN = ±3.3V can be measured. The code range that represents voltages with |VIN| > 3.3V is not used in this case.

From programming point you get a value between 0 and 2**16-1, where 4.096V has the full scale of 0x7FFF. 3.3V is represented by a lower digital value.

Creation:

```
rads=Remote_WRADS1115(rs,0,address=0x48,gain=1)     [single.mode]
rads1=Remote_WRADS1115(rs,0,1,address=0x48)         [differential mode]
```

You don't need to set the mode. ADS1115 recognizes it automatically by the number of channels.

rads.value is 2-tuple with digital value and voltage.

5.4 Remote_Kontext

As design pattern this is a Compositum of objects with a value property and setClientIdent() and getClientIdent() methods. The idea is to freeze the values of all devices as a state of the system by state=rctxt.value. Then you verify whether the state has to be changed. For each device you make the change. Of more interest is when the system has only a finite number of states and you can decide to change from one state to the other. This is the case when you have only a finite number of digital outputs in the system. E.g. for a control of a heating with two pumps and a heating burner you need at maximum 8 states. The MCP23x17 support this concept. You can change all digital outputs at the same time because they are in one register and changeable at the same time.

rctxt =Remote_Kontext(rdev0,…,rdevn)
rdevo, …,rdevn may be a Remote_DigitalDevice or RMCButton and RMCLED of MCP23x17.

Properties:	value

methods:	getClientIdent(),
	setClientIdent(val:str)
	addComponent(*components:Remote_DigitalDevice | RMCButton | RMCLED)
	popComponent(component)
	getChild(index:int)

iterables:	rctxt._childs=[]
	
	rctxt._value={} (keys().values(), items())
	rctxt._basis_elements=[]

for any child you find the stocked value by:

```
	freezeValue= rctxt._value[child.getClientIdent()]
e.g.	freeze_Value=rctxt._value[‘rl'}
```

see the example in controller.py

6. remoteio_devices.py

6.1. Overview of realized gpiozero Devices

These are the remoteio counterparts of gpiozero devices. Not all gpiozero devices are supported.

The following devices are supported. For use look to the examples in controller. Explications are only given when the functionality in remoteio is extended.

Remote_AngularServo
Remote_Button
Remote_DistanceSensor
Remote_LED
Remote_LEDBarGraph
Remote_LEDBoard
Remote_LEDCompositum
Remote_LightSensor
Remote_LineSensor
Remote_MCP3208
Remote_MotionSensor
Remote_Motor
Remote_PWMLED
Remote_PhaseEnableMotor
Remote_RGBLED
Remote_Servo
Remote_TonalBuzzer

Detailed examples are found in controller.py or at the end of the corresponding .py-files

6.2. Special explications

6.2.1 Remoteio_LightSensor

Even with the experimental setup as described in the API of gpiozero it was not possible to generate some reaction of the gpiozero LightSensor code.

As an alternative the proceeding described in the example of MCP3208 is recommended (look in remote_mcp3208.py)

6.2.2 Remote_RotaryEncoder

Of interest is the use of the helper function shortetsWay() for counting clockwise or counterclockwise. The sample rate is so short that a human being can only turn around the shortest way. By this way counting problems in the case wrap==True are solved.

The use of the push button SW of the rotary encoder is supported. There exists an internal support by the method activateSW(self,pinNr,pull_up=True) but it is also programmable independent of the internal solution. See the examples. Further there is a reset_counter method useable as when_pressed function, so that in the case of pressing SW an internal counter is reset to zero.

6.2.3. Remote_LEDCompositum

This is not a gpiozero device. It is noted in the list of gpiozero devices because it was firstly intended only to work with gpiozero devices. This is a class that only works on the client side. It is made following the design-pattern of compositum. All devices with the properties on,off,toggle,blink, value are allowed.

On compositum level we have on(*args,**kwargs), off(*args), blink(*args,**kwargs),

toggle(*args), pulse(*args,**kwargs) when possible. On basis_element-level it reduces

to ...(**kwargs).

Further Remote_LedCompositum has a close-function. This compositum is not restricted to LEDs.

It support all devices with the above properties that are registered by RemoteSupervisor. So RMCPLED and RMCPButton of MCP23x17 are also supported.

7. remoteio_client.py

Classes of interest are RemoteSupervisor, RemoteServer, RemoteDigitalDevice.

They are to found in all device-classes presented before. Controller.py is written as a collection of examples but is also a template for working with remoteio on client side.

8. remoteio_server.py

The method run-server is of interest for the user. The code server_start.py is universal for the start of the server side.