Module `htpio`

Hardware Test Platform (HTP) Remote I/O Classes and Modules.

HTPIO is an executable package, which allows the user to test or use some of its library functionality from the command line.

Set value using telnet:

```
htpio -p telnet set --host 192.168.199.31 --pin 14 --val 1
```

Get value using socket:

```
htpio -p socket get --host 192.168.199.31 --pin 14
```

Lock/Unlock pin:

```
htpio -p telnet lock --host 192.168.199.31 --pin 14
htpio -p socket unlock --host 192.168.199.31 --pin 14
```

Sub-modules

- htpio.bridge
- htpio.decorators
- htpio.exceptions
- htpio.socketio
- htpio.telnetio

Module `htpio.bridge`

This module provides the connection between the client application and the remote i/o
implementation.

Example :

```
import time
import htpio.bridge as bridge
import htpio.telnetio as telnetio
import htpio.socketio as socketio

device = telnetio.RaspberryPi('192.168.199.31')
gpio = bridge.RemoteAccess(device)

gpio.lock(14)
gpio.configure(14, gpio.OUTPUT)

gpio.set(14, 1)
time.sleep(1)
gpio.set(14, 0)

gpio.unlock(14)
gpio.deconfigure(14)
gpio.logout()

device = socketio.RaspberryPi('192.168.199.31')
gpio = bridge.RemoteAccess(device)

gpio.lock(15)
gpio.configure(15, gpio.OUTPUT)

gpio.set(15, 1)
time.sleep(1)
gpio.set(15, 0)

gpio.unlock(15)
gpio.deconfigure(15)
gpio.logout()
```

Classes

Class `Device`

> `class Device(*args, **kwargs)`

Define the implementor's interface from the bridge pattern. This interface provides homogeneous interface for all remote I/O libraries.

Typically the implementor interface provides only primitive operations, and abstraction defines higher-level operations based on these primitives.

Descendants

- htpio.socketio.RaspberryPi
- htpio.telnetio.RaspberryPi

Class variables

Variable `INPUT`

Pin input mode

Variable `OUTPUT`

Pin output mode

Methods

Method `configure`

```
def configure(self, pin, direction)
```

Abstract method

Method `deconfigure`

```
def deconfigure(self, pin)
```

Abstract method

Method `get`

```
def get(self, pin)
```

Abstract method

Method `isconfigured`

```
def isconfigured(self, pin)
```

Abstract method

Method `islocked`

```
def islocked(self, pin)
```

Abstract method

Method `lock`

```
def lock(self, pin)
```

Abstract method

Method `login`

```
def login(self)
```

Abstract method

Method `logout`

```
def logout(self)
```

Abstract method

Method `reset`

```
def reset(self, pin)
```

Abstract method

Method `set`

```
def set(self, pin, value)
```

Abstract method

Method `unlock`

```
def unlock(self, pin)
```

Abstract method

Class `RemoteAccess`

```
class RemoteAccess(device_api)
```

Define the abstraction's interface from the bridge pattern used by the client application.

Typically the implementor interface provides only primitive operations, and abstraction defines higher-level operations based on these primitives.

Methods

Method `configure`

> `def configure(self, pin, direction)`

Configure pin with direction if it is free

Example:

```
from htpio.bridge import RemoteAccess
from htpio import telnetio, socketio

remoteio = socketio.RaspberryPi('192.168.199.31')
led = RemoteAccess(remoteio)
led.configure(14, led.OUTPUT)
led.set(14, 1)
...
```

Method `deconfigure`

> `def deconfigure(self, pin)`

Deconfigure pin if it belongs to my session

Example:

```
from htpio.bridge import RemoteAccess
from htpio import telnetio, socketio

remoteio = socketio.RaspberryPi('192.168.199.31')
led = RemoteAccess(remoteio)
led.deconfigure(14)
...
```

Method `get`

> `def get(self, pin)`

Get pin value

Example:

```
from htpio.bridge import RemoteAccess
from htpio import telnetio, socketio

remoteio = socketio.RaspberryPi('192.168.199.31')
led = RemoteAccess(remoteio)
led.configure(14, led.OUTPUT)
led.set(14, 1)
print(led.get(14))
...
```

Method `isconfigured`

> `def isconfigured(self, pin)`

Check if pin is configured

Example:

```
from htpio.bridge import RemoteAccess
from htpio import telnetio, socketio

remoteio = socketio.RaspberryPi('192.168.199.31')
led = RemoteAccess(remoteio)
led.isconfigured(14)
...
```

Method `islocked`

> `def islocked(self, pin)`

Check if pin is locked by another process

Example:

```
from htpio.bridge import RemoteAccess
from htpio import telnetio, socketio

remoteio = socketio.RaspberryPi('192.168.199.31')
led = RemoteAccess(remoteio)
led.lock(14)
if led.islocked(14):
    print("LED is locked!!!")
```

```
led.logout()
...
```

Method `lock`

> `def lock(self, pin)`

Lock pin if it is not used

Example:

```
from htpio.bridge import RemoteAccess
from htpio import telnetio, socketio

remoteio = socketio.RaspberryPi('192.168.199.31')
led = RemoteAccess(remoteio)
led.lock(14)
...
```

Method `login`

> `def login(self)`

Registers to remote device

Example:

```

from htpio.bridge import RemoteAccess
from htpio import telnetio, socketio

remoteio = socketio.RaspberryPi('192.168.199.31')
led = RemoteAccess(remoteio)
led.login()
...
```

Method `logout`

> `def logout(self)`

Unregisters from remote device

Example:

```
from htpio.bridge import RemoteAccess
from htpio import telnetio, socketio

remoteio = socketio.RaspberryPi('192.168.199.31')
led = RemoteAccess(remoteio)
led.logout()
...
```

Method `reset`

> `def reset(self, pin)`

Remove pin lock forcefully

Example:

```
from htpio.bridge import RemoteAccess
from htpio import telnetio, socketio

remoteio = socketio.RaspberryPi('192.168.199.31')
led = RemoteAccess(remoteio)
led.lock(14)
print(led.islocked(14))
led.reset(14)
print(led.islocked(14))
...
```

Method `set`

> `def set(self, pin, value)`

Write value to pin

Example:

```
from htpio.bridge import RemoteAccess
from htpio import telnetio, socketio

remoteio = socketio.RaspberryPi('192.168.199.31')
led = RemoteAccess(remoteio)
led.configure(14, led.OUTPUT)
led.set(14, 1)
```

```
print(led.get(14))
...
```

Method `unlock`

> `def unlock(self, pin)`

Unlock pin if it belongs to my session

Example:

```
from htpio.bridge import RemoteAccess
from htpio import telnetio, socketio

remoteio = socketio.RaspberryPi('192.168.199.31')
led = RemoteAccess(remoteio)
led.unlock(14)
...
```

Module `htpio.decorators`

This module implement various decorators such as function calls and exception logging.
Decorators are used to implement additional functionality to a method or function without
changing its name.

Currently the following decorators are implemented :

- @countcalls - Counts the number of function calls
- @logger(filename) - Logs function exceptions to a give file

Example:

```
    from htpio.decorators import Countcalls, Logger

    @Countcalls
    def f():
        pass

    count = 0
```

```
    for i in range(100):
        count = i + 1
        f()

    print(f.count(), count)
```

```
    @Countcalls
    @Logger(LOGFILE)
    def f():
        return 1 / 0

    # Function under test
    raised = False
    try:
        f()
    except:
        raised = True
        logger.log.info("Number of calls==[{0}]".format(f.count(
```

Classes

Class `Countcalls`

> `class Countcalls(f)`

Decorator that keeps track of the number of times a function is called.
https://wiki.python.org/moin/PythonDecoratorLibrary

Example:

```
from htpio.decorators import Countcalls

@Countcalls
def f():
    pass

@Countcalls
def g():
    pass

count = 0
for i in range(100):
    count = i + 1
    f()
```

```
    g()

print(f.count())  # Number of calls for funciton f
print(g.count())  # Number of calls for function g
print(Countcalls.counts()) # Dump number of calls for f and g
```

Class variables

Variable `instances`

Stores the function names and number of calls

Static methods

`Method counts`

> `def counts()`

Return a dictionary for all registered functions is keys and the number of function calls as values.

Methods

Method `count`

> `def count(self)`

Return the number of times the function was called.

Class `Logger`

> `class Logger(logfile)`

Decorator that logs exceptions into a given file.

Example:

```
from htpio.decorators import Logger

@Logger('c:\test.log')
def f():
    return 1 / 0

### Function under test
raised = False
```

```
try:
    f()
except:
    raised = True
    logger.log.info("Number of calls==[{0}]".format(f.count()))
```

Class variables

Variable `log`

Instance of a logging object

Variable `logfile`

Log file use to store the logs

Module `htpio.exceptions`

This module implements all exception classes used by htpio.

Classes

Class `CannotConnectToTarget`

> `class CannotConnectToTarget(*args, **kwargs)`

Raised the object cannot connect to the specified [target:socket].

Ancestors (in MRO)

- builtins.Exception
- builtins.BaseException

Class `CannotCreateLockDirectory`

> `class CannotCreateLockDirectory(*args, **kwargs)`

Raised when the lockfolder cannot be created.

Ancestors (in MRO)

- builtins.Exception

- builtins.BaseException

Class `CannotMountRamDisk`

```
class CannotMountRamDisk(*args, **kwargs)
```

Raised when the ram disk cannot be mapped to the lock folder.

Ancestors (in MRO)

- builtins.Exception
- builtins.BaseException

Class `InvalidLoginDetails`

```
class InvalidLoginDetails(*args, **kwargs)
```

Raised when user or password provided not valid.

Ancestors (in MRO)

- builtins.Exception
- builtins.BaseException

Class `PinLockedByOtherProcess`

```
class PinLockedByOtherProcess(lock_owner=None,
my_session=None)
```

Raised when the gpio pin locked by another htpio process.

Ancestors (in MRO)

- builtins.Exception
- builtins.BaseException

Module `htpio.socketio`

This is a socket i/o control module wrapping the python pigpio client.
PIGPIO is a client/server
library using sockets for remote control of the general purpose input outputs (GPIO).

Example:

```
import htpio.socketio as socketio

r = socketio.RaspberryPi(host = '192.168.199.31',
                         port = 8888)

try:
    r.lock(14)
except:
    r.logout()
    raise

r.configure(14, socketio.RaspberryPi.OUTPUT)
r.set(14, 1)

print(r.get(14))

r.unlock(14)
r.deconfigure(14)
```

Classes

Class `RaspberryPi`

> `class RaspberryPi(host=None, port=8888)`

This class allows users to connect and control the GPIO
on a Raspberry remotely by using and extending the pigpio library.

Configuration

Enable Remote GPIO

```
sudo raspi-config
Menu : Interfacing Options -> Remote GPIO
```

Enable pigpiod to start on boot

```
sudo systemctl enable pigpiod
```

Class attributes

```
LOCK_DIR        : Location of lock files (ramdisk)
MAX_GPIO        : Maximum number of gpio locks ( from 0 .. MAX)
INPUT           : GPIO is digital input
OUTPUT          : GPIO is digital output
EOF             : End of file
```

Instance attributes

```
host            : ip v4 address
port            : port (from 0 to 65535)
```

Public methods

```
islocked(gpio)              : Check if the gpio is locked
lock(gpio)                  : Locks the gpio
unlock(gpio)                : Unlocks the gpio
isconfigured(gpio)          : Checks if the gpio is configured
configure(gpio, direction)  : Configures the gpio with direction
deconfigure(gpio)           : Restores the default configuration
get(gpio)                   : Reads the gpio status
set(gpio, value)            : Writes new value to gpio
login()                     : Registers to the remote system
logout()                    : Unregisters from the remote system
reset()                     : Deletes all configuration data
```

Constructor

When called with host address, the constructor creates
a telnet connection and performs automatic login (default=8888).

```
import htpio.socketio as socketio

t = socketio.RaspberryPi('192.168.199.31')
```

Ancestors (in MRO)

- pigpio.pi
- htpio.bridge.Device

Class variables

Variable `EOF`

Symbol for end of transmission

Variable `LOCKDIR`

Lock files directory

Variable `MAX_GPIO`

Maximum number of GPIOs

Methods

Method `configure`

> `def configure(self, pin, direction)`

Configures the gpio for input or output operation.

Example:

```
import htpio.socketio as socketio

led = socketio.RaspberryPi('192.168.199.31')

led.configure(14, socketio.RaspberryPi.OUTPUT)
...
```

Method `deconfigure`

> `def deconfigure(self, pin)`

Removes the gpio configuration.

Example:

```
import htpio.socketio as socketio

led = socketio.RaspberryPi('192.168.199.31')
```

```
led.deconfigure(14)
...
```

Method `get`

> `def get(self, pin)`

Reads the value of the gpio.

Example:

```
import htpio.socketio as socketio

led = socketio.RaspberryPi('192.168.199.31')

led.get(14)
...
```

Method `isconfigured`

> `def isconfigured(self, pin)`

Check if gpio is configured by comparing the current direction configuration with the default one (INPUT).

Example:

```
import htpio.socketio as socketio

led = socketio.RaspberryPi('192.168.199.31')

led.configure(14, socketio.RaspberryPi.OUTPUT)
if led.isconfigured(14):
    print("LED is configured!!!")
...
```

Method `islocked`

> `def islocked(self, pin)`

Opens the gpio lock file and checks it size. If the size is more than zero then the gpio is considered to be locked by another process.

Example:

```
import htpio.socketio as socketio

led = socketio.RaspberryPi('192.168.199.31')
led.lock(14)
if led.islocked(14) :
    print("LED is locked!!!")
...
```

Method `lock`

> `def lock(self, pin)`

Lock the given pin by creating a lock file and writing the current session id into it.

Example:

```
import htpio.socketio as socketio

led = socketio.RaspberryPi('192.168.199.31')
led.lock(14)
...
```

Method `login`

> `def login(self)`

Registers device for use with target device by checking for an established connection and creating all gpio lock files.

Example:

```
import htpio.socketio as socketio

led = socketio.RaspberryPi()
led.host = '192.168.199.31'
led.port = 8888
led.login()
...
```

Method `logout`

> `def logout(self)`

Unregisters device from remote device.

Example:

```
import htpio.socketio as socketio

led = socketio.RaspberryPi('192.168.199.31')

led.logout()

...
```

Method `reset`

> `def reset(self, pin)`

Deletes the content of all generated lock files.

Example:

```
import htpio.socketio as socketio

led = socketio.RaspberryPi('192.168.199.31')

led.reset(14)

...
```

Method `set`

> `def set(self, pin, value)`

Writes the value to the gpio.

Example:

```
import htpio.socketio as socketio

led = socketio.RaspberryPi('192.168.199.31')

led.configure(14, socketio.RaspberryPi.OUTPUT)
led.set(14, 1)
...
```

Method `unlock`

> `def unlock(self, pin)`

Unlock the pin by truncating the size of the lock file to zero.

Example:

```
import htpio.socketio as socketio

led = socketio.RaspberryPi('192.168.199.31')
led.unlock(14)
...
```

Module `htpio.telnetio`

This is a telnet client i/o control module.
Telnet is a client/server text-oriented communication protocol using a virtual terminal
connection and operates over TCP. It provides a command-line interface to the operating system
on a remote host.

Example:

```
import htpio.telnetio as telnetio

r = telnetio.RaspberryPi(host = '192.168.199.31',
                         port = 23,
                         user = 'pi',
                         password = 'raspberry')

try:
    r.lock(14)
except:
    r.logout()
    raise

r.configure(14, telnetio.RaspberryPi.OUTPUT)
r.set(14, 1)

print(r.get(14))
```

```
r.unlock(14)
r.deconfigure(14)
```

Classes

Class `RaspberryPi`

> `class RaspberryPi(host=None, port=23, user='htp', password='sokotnar', login_prompt='login:', password_prompt='Password:', shell_prompt='$', timeout=20)`

RaspberryPi telnet remote i/o access class.

This class provides basic functionality for controlling the GPIO by using a remote telnet session and sending commands to the operating system.

Configuration

The user should be part of the sudo and gpio group and the visudo file should be configured for no password prompt when using the sudo. This is accomplished by executing the following commands :

```
sudo adduser htp
sudo usermod -a -G gpio htp
sudo usermod -a -G sudo htp

sudo visudo
    (add line) htp     ALL=(ALL) NOPASSWD: ALL
    (add line) %gpio   ALL=(ALL) NOPASSWD: ALL
```

Class attributes

```
LOCK_DIR         Location of lock files (ramdisk)
MAX_GPIO         Maximum number of gpio locks ( from 0 .. MAX)
INPUT            GPIO is digital input
OUTPUT           GPIO is digital output
EOF              End of file
```

Instance attributes

```
host            : ip v4 address
port            : port (from 0 to 65535)
user            : user
password        : user password
login_prompt    : token to detect when login is expected
password_prompt : token to detect when password is expected
shell_prompt    : token to detect return after command execution
timeout         : timeout time in case of connectivity problems
```

Public methods

```
islocked(gpio)              : Check if the gpio is locked
lock(gpio)                  : Locks the gpio
unlock(gpio)                : Unlocks the gpio
isconfigured(gpio)          : Checks if the gpio is configured
configure(gpio, direction)  : Configures the gpio with direction
deconfigure(gpio)           : Restores the default configuration
get(gpio)                   : Reads the gpio status
set(gpio, value)            : Writes new value to gpio
login()                     : Registers to the remote system
logout()                    : Unregisters from the remote system
reset()                     : Deletes all configuration data
```

Constructor

When called with host address, the constructor creates
a telnet connection and performs automatic login:

```
import htpio.telnetio as telnetio

t = telnetio.RaspberryPi('192.168.199.31')
```

When called without host address, the constructor creates
an unconnected instance. In this case the object might be
configured by using the instance attributes:

```
import htpio.telnetio as telnetio

t = telnetio.RaspberryPi()
```

```
t.host = '192.168.199.31'
t.port = 23
t.user = 'user'
t.password = 'password'
t.login_prompt = 'login:'
t.password_prompt = 'Password:'
t.shell_prompt = '$'
t.timeout = 20
```

After the configuration above the user is required to use the following statements:

```
t.open(host, port)
t.login()
```

Ancestors (in MRO)

- telnetlib.Telnet
- htpio.bridge.Device

Class variables

Variable `EOF`

Character used to mark end of line transmission

Variable `LOCKDIR`

Directory containing the lock files

Variable `MAX_GPIO`

Maximum number of gpio lock files

Variable `RAMDISK`

Label for the ramdisk mounted on the lock directory

Methods

Method `configure`

> `def configure(self, pin, direction)`

Create the sysfs virtual folder for gpio manipulation.

Shell :

```
echo 14 | tee /sys/class/gpio/export
```

Example:

```
import htpio.telnetio as telnetio

led = telnetio.RaspberryPi('192.168.199.31')

led.configure(14, telnetio.RaspberryPi.OUTPUT)
...
```

Method `deconfigure`

> `def deconfigure(self, pin)`

Delete the sysfs virtual folder for pin manipulation.

Shell :

```
echo 14 | tee /sys/class/gpio/unexport
```

Example:

```
import htpio.telnetio as telnetio

led = telnetio.RaspberryPi('192.168.199.31')

led.deconfigure(14)
...
```

Method `get`

> `def get(self, pin)`

Gets the current status of the given pin by reading the system file.

Shell :

```
cat /sys/class/gpio/gpio14/value
```

Example:

```
import htpio.telnetio as telnetio

led = telnetio.RaspberryPi('192.168.199.31')

led.get(14)
...
```

Method `isconfigured`

> `def isconfigured(self, pin)`

This function checks if the sysfs folder /sys/class/gpio/gpioNN exists.

Shell:

```
find /sys/class/gpio -name gpio14
```

Example:

```
import htpio.telnetio as telnetio

led = telnetio.RaspberryPi('192.168.199.31')

led.configure(14, telnetio.RaspberryPi.OUTPUT)
if led.isconfigured(14):
    print("LED is configured!!!")
...
```

Method `islocked`

> `def islocked(self, pin)`

Check if pin is already locked by an object of this class and returns the identification number of the locking process.

Shell:

```
sudo cat /tmp/htp/locks/gpio14
```

Example:

```
import htpio.telnetio as telnetio

led = telnetio.RaspberryPi('192.168.199.31')
led.lock(14)
if led.islocked(14) :
    print("LED is locked!!!")
...
```

Method `lock`

> `def lock(self, pin)`

Throws an exception **PinLockedByOtherProcess** or opens the gpio lock file and writes the session id in it.

Shell:

```
echo 1921681993055555 | sudo tee /tmp/htp/gpio14
```

Example:

```
import htpio.telnetio as telnetio

led = telnetio.RaspberryPi('192.168.199.31')
led.lock(14)
...
```

Method `login`

> `def login(self)`

Performs an automatic login to the remote system using the supplied user and password as instance properties.

!!! Use only when creating an unconnected instance !!!

Example:

```
import htpio.telnetio as telnetio

led = telnetio.RaspberryPi()
```

```
led.host = '192.168.199.31'
led.port = 23
led.user = 'htpio'
led.password = 'sokotnar'

led.open(host, port)
led.login()

...
```

Method `logout`

> `def logout(self)`

Closes the connection to the target.

Example:

```
import htpio.telnetio as telnetio

led = telnetio.RaspberryPi('192.168.199.31')

led.logout()

...
```

Method `reset`

> `def reset(self, pin)`

Deletes lock file for pin.

Example:

```
import htpio.telnetio as telnetio

led = telnetio.RaspberryPi('192.168.199.31')

led.reset(14)

...
```

Method `set`

> `def set(self, pin, value)`

Sets the current value of the given pin by using the system file.

Shell:

```
echo 1 | sudo tee /sys/class/gpio/gpio14
```

Example:

```
import htpio.telnetio as telnetio

led = telnetio.RaspberryPi('192.168.199.31')

led.configure(14, telnetio.RaspberryPi.OUTPUT)
led.set(14, 1)
...
```

Method `unlock`

> `def unlock(self, pin)`

Unlocks the gpio by deleting the content of the lock file.

Shell:

```
sudo truncate --size=0 /tmp/htp/locks/gpio14
```

Example:

```
import htpio.telnetio as telnetio

led = telnetio.RaspberryPi('192.168.199.31')
led.unlock(14)
...
```
