SCHLEIFENBAUER
LIVING FOR THE POWER TO DELIVER

HLAPI Tutorial

Version 1.1.3

1. About

The Schleifenbauer High-Level Python API allows mass configuration and visualization of Schleifenbauer Products through ethernet. The HLAPI dynamically communicates with Schleifenbauer devices using both HTTP and IPAPI.

2. Setup

2.1 Dependencies

In order to use the high-level API, Python version 3.5 or higher must be installed on your system.

2.2 Installation

Download the HLAPI from http://documentation.schleifenbauer.eu and unzip the downloaded archive.

2.3 Duality

The high-level API can be used in two ways: from the command line and as a python package. The command line usage is limited to simple reading and writing of data to a defined list of targets. When importing the HLAPI from a python script you will have full control over the API at the cost of complexity.

3. Command line usage

3.1 Preparation

Change your current working directory to the folder above *hlapi*.

```
$ cd path/to/folder/above/hlapi
$ ls
hlapi
```

HLAPI needs to be executed as a Python module. The command line syntax is:

```
$ python3 -m hlapi.hlapi
    -t /path/to/targets.json [required]
    -r "register1,register2,..."
    -w "register1=value1,register2=value2,..."
    -s, --pretty, --debug
```

Note that the *-m* flag tells Python to execute the file *hlapi* in the package *hlapi* as a module.

SCHLEIFENBAUER
LIVING FOR THE POWER TO DELIVER

3.2 Command line options

-t *Targets JSON file (required)*
This parameter must point to a file containing the target devices with corresponding credentials. Example file contents:

```
{
      "192.168.1.*": {
            "webapi_port": 80,
            "webapi_user": "username",
            "webapi_pass": "password",
            "ipapi_key": "1234567890123456"
      },
      "192.168.2.200": {
            "webapi_port": 80,
            "webapi_user": "username",
            "webapi_pass": "password",
            "ipapi_key": "1234567890123456"
      },
      "192.168.3.111#15": {
            "webapi_port": 80,
            "webapi_user": "username",
            "webapi_pass": "password",
            "ipapi_key": "1234567890123456"
      }
}
```

Each first-level key represents an interface which could either be: an entire subnet (*192.168.1.** in the example above), an interface including all devices connected to it which would be a bridge/hybrid PDU or gateway (direct IP address, 192.168.2.200 in the example above*)* or a specific unit in a ring of Schleifenbauer devices (*192.168.3.111#15* in the example above).

HLAPI automatically filters out duplicate entries such as *192.168.1.** and *192.168.1.20*. Subnet scans are prioritized, interfaces after that and direct devices in the last place.

Note that when scanning a subnet, the given credentials will be used for all found devices.

-r *Read*
This parameter requires a comma-separated list of register mnemonics (refer to the SPDM).
`Example: python3 -m hlapi.hlapi -t exampletargets.json -r "idfwvs,idsnbr,idchip"`

-w *Write*
This parameter requires a comma-separated list of register mnemonics (refer to the SPDM) and associated write values using the 'equals to' sign (=).

```
Example: python3 -m hlapi.hlapi -t exampletargets.json -w
"stdvnm=some_name,stdvlc=some_location"
```

You are required to use either -r or -w, both cannot be used at the same time.

-s Silent (optional)
Only prints the final result in JSON to the console.

--pretty (optional)
Pretty-prints the final result in JSON to the console.

--debug (optional)
Enables debug mode, verbose output will be printed to the console.

4. Python usage

4.1 Preparation

To use the high-level API to it's full extent, it's best to import the HLAPI from a python script.
Get started by creating a python script in the same folder as the *hlapi* folder.

To execute the script, open a command prompt and change your current working directory to the folder containing *demo.py* and the *hlapi* folder, call Python 3.5 and pass *demo.py* as an argument.

In order to use any functionality of the HLAPI, a main *HLAPI()* instance needs to be created.

```
from hlapi.hlapi import HLAPI

hlapi = HLAPI(debug=False)
```

The *HLAPI()* instance keeps track of settings such as timeouts, cache and configuration parameters.

4.2 Device identification

4.2.1 Specific interface(s)

In order to perform operations on devices, Device objects need to be instantiated. Let's say you want to read some identification info from all devices behind (and including) a bridge PDU.

The *DeviceManager* class takes care of interface scanning and device identification for you.

For example, if the target interface has IP address *192.168.1.100*, DeviceManager will need to scan this interface's ring for Schleifenbauer devices. Let's start by scanning and identifying devices using *DeviceManager.loadInterfaces(dict)*. This function takes a python dictionary as an argument, the keys can either be IP addresses or IP#UNIT combinations. If an IP is given, all units behind that IP will respond, otherwise only the specified devices will respond. Per interface, the values *webapi_port*, *webapi_user*, *webapi_pass* and *ipapi_key* will need to be passed in order to establish a connection.

```
from hlapi.hlapi import HLAPI
from hlapi.DeviceManager import DeviceManager

hlapi = HLAPI(debug=False)
deviceManager = DeviceManager(hlapi)

interfaces = {
            "192.168.9.100": {
                  "webapi_port": 80,
                  "webapi_user": "power",
                  "webapi_pass": "power",
                  "ipapi_key": "0000000000000000"
            }
}

deviceManager.loadInterfaces(interfaces)

for device in deviceManager.devices:
      print(device)
```

After running this script, our DeviceManager instance has a *'devices'* variable that contains a list of Device objects. Each one of these objects represents a physical Schleifenbauer device.

```
$ python3 device_identification_demo.py
<hlapi.devices.Devices.hPDU object at 0x10c98bac8>
<hlapi.devices.Devices.hPDU object at 0x103152ba8>
<hlapi.devices.Devices.cPDU object at 0x10c80dcf8>
<hlapi.devices.Devices.hPDU object at 0x10c98beb8>
<hlapi.devices.Devices.cPDU object at 0x10c98ba20>
```

4.2.2 Network scanning (subnet)

HLAPI has a built-in network scanner making it easy to detect Schleifenbauer devices on your network. In this example we will be scanning the subnet 192.168.1.0 until 192.168.1.255:

```
import time

from hlapi.hlapi import HLAPI
from hlapi.NetworkScanner import NetworkScanner
from hlapi.DeviceManager import DeviceManager

hlapi = HLAPI(debug=False)

subnet = "192.168.9.*" # * means scan from .0 to .255
http_port = 80
webapi_user = "power"

# Start scanning
networkScanner = NetworkScanner(hlapi, subnet, http_port,
webapi_user)
networkScanner.startScan()

# Wait until scanner is done, more info on progress in section 4.4
while networkScanner.progress.isRunning():
      time.sleep(1)

# Print the result
print(networkScanner.result)
```

```
$ python3 network_scanning_demo.py
```

After about 25 seconds, a list of all IP addresses that appear to be Schleifenbauer devices is returned.
This list can be fed into DeviceManager (as explained in section 4.2.1) to scan databus rings and identify all all associated devices.

4.3 Synchronously reading/writing data

In order to read or write data from/to devices, there are a few options. Let's go over them in ascending level of usage complexity.

Read	Write
→ *MultiReadWrite.readSingle()* → *MultiReadWrite.readAll()* → Abstract*Device.read()* → *Communicator.read()* → [WEB/IP]API*Protocol.read()*	→ *MultiReadWrite.writeSingle()* → *MultiReadWrite.writeAll()* → Abstract*Device.write()* → *Communicator.write()* → [WEB/IP]API*Protocol.write()*

It makes sense to go over each way of accessing data from high to low level.
Note: see section 5.1 for a detailed schematic of how the HLAPI is structured.

4.3.1 Read/write a single register using MultiReadWrite.readSingle()

MultiReadWrite.readSingle() allows you to read a single registers from a set of devices. Let's try to read the firmware version of all devices behind (and including) interface 192.168.1.100:

```python
from hlapi.hlapi import HLAPI
from hlapi.DeviceManager import DeviceManager
from hlapi.managers.MultiReadWrite import MultiReadWrite

interfaces = {
      "192.168.9.100": {
            "webapi_port": 80,
            "webapi_user": "power",
            "webapi_pass": "power",
            "ipapi_key": "0000000000000000"
      }
}

hlapi = HLAPI(debug=False)

# identify devices
deviceManager = DeviceManager(hlapi)
```

On execution, this gives us a list of all identified devices and their associated firmware versions:

```
$ python3 read_write_registers_managers_demo.py
192.168.1.100#1 244
192.168.1.100#2 244
192.168.1.100#3 150
192.168.1.100#4 150
192.168.1.100#5 244
192.168.1.100#6 244
```

4.3.2 Read/write multiple registers using *MultiReadWrite.readAll()*

You might want to read more than a single register from a selection of devices. *MultiReadWrite.readAll()* is a powerful function, it allows reading multiple registers from multiple devices either synchronously or asynchronously.

This function takes a list of register mnemonics as input. Refer to the SPDM for documentation on the registers available.

For this example we'll try to read 3 registers from 5 devices: firmware version, serial number and the devices' maximum rated load. The register mnemonics for these registers are *idfwvs*, *idsnbr* and *cfamps* respectively.

```
from hlapi.hlapi import HLAPI
from hlapi.DeviceManager import DeviceManager
from hlapi.managers.MultiReadWrite import MultiReadWrite

interfaces = {
      "192.168.9.100": {
            "webapi_port": 80,
            "webapi_user": "power",
            "webapi_pass": "power",
            "ipapi_key": "0000000000000000"
      }
}

mnemonics = ['idfwvs', 'idsnbr', 'cfamps']

hlapi = HLAPI(debug=False)

# identify devices
```

Upon execution, this gives us the following result:

```
$ python3 abstract_manager_readall_demo.py
192.168.1.100#1
        idfwvs 244
        idsnbr SVNL00036483
        cfamps 16
192.168.1.100#2
        idfwvs 244
        idsnbr SVNL00042553
        cfamps 32
192.168.1.100#3
        idfwvs 150
        idsnbr SVNL00026870
        cfamps 32
192.168.1.100#4
        idfwvs 150
        idsnbr SVNL00018449
        cfamps 32
192.168.1.100#5
        idfwvs 244
        idsnbr SVNL00042856
        cfamps 32
```

4.3.3 Direct device read/write

Each Device object has its own *read()* and *write()* functions. Retrieved data is cached per Device object. (A note on caching: when reading data from a device which as been read previously, the cached data will be returned if this data has not yet expired (depending on the config value, 1000 seconds by default). You can disable caching globally by setting the *cache_expire* config value to 0 or by passing *cache=False* as an argument to *Device.read())*

The *read()* function takes a *readValue* and a *readType* as arguments. *readType* must be either ‘single’ or ‘group’. The *readValue* must be a valid register mnemonic if *readType* is ‘single’ or an SPDM group name if *readType* is ‘group’.

When a single register is read, the entire group that this register belongs to will automatically be read for caching purposes since Schleifenbauer devices respond just as quickly to a group read as a single register read.

To demonstrate direct register reading, writing and caching we will first read a single register (device name, *stdvnm*) twice, then set a new device name and read the register again.

```python
from time import time

from hlapi.hlapi import HLAPI
from hlapi.DeviceManager import DeviceManager
from hlapi.managers.MultiReadWrite import MultiReadWrite

interfaces = {
      "192.168.9.100": {
            "webapi_port": 80,
            "webapi_user": "power",
            "webapi_pass": "power",
            "ipapi_key": "0000000000000000"
      }
}

hlapi = HLAPI(debug=False)

deviceManager = DeviceManager(hlapi)
deviceManager.loadInterfaces(interfaces)
# take only the first device
targetDevice = deviceManager.devices[0]

start_time = time()
print(targetDevice.read('stdvnm', 'single'))
print("Elapsed: {0:.3f}".format(time()-start_time), "s")

start_time = time()
print(targetDevice.read('stdvnm', 'single'))
print("Elapsed: {0:.3f}".format(time()-start_time), "s")

targetDevice.write('stdvnm', 'single', 'new_name')

print(targetDevice.read('stdvnm', 'single'))
```

Output:

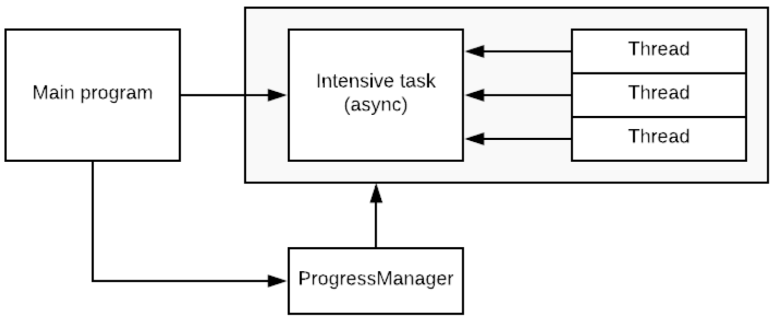
```
$ python3 direct_read_write_demo.py
{'stdvnm': 'foobar'}
Elapsed: 0.664 s
{'stdvnm': 'foobar'}
Elapsed: 0.000 s
{'stdvnm': 'new_name'}
```

As can be seen, the first read takes a little more than half a second, the second time this register is returned instantly from cache and after writing a new value, the cache is reset.

4.4 Asynchronous usage and *ProgressManagers*

When communicating with many devices behind different interfaces at the same time, execution times will increase. You might want the API to read or write data while your application does something else in the meantime. Luckily, most functions such as mass reading, writing, identification or network scanning can be called asynchronously.
The asynchronous functions throughout the HLAPI all use a so-called ProgressManager. This is an object that keeps track of a background operation's status, progress and associated threads.

Here's how a ProgressManager works:



A ProgressManager will keep track of a task's progress, target, percentage, status, other (child) ProgressManagers and associated threads.

- The state can either be None, running, finished, aborted or error. A ProgressManager is initialized once a target value has been set. The ProgressManager's internal state will change from *None* it's defined 'run state'. The method *waitForInit()* blocks until a target has been set.
- Progress can be set or added using *setProgress()* and *addProgress()* respectively. The same goes for the ProgressManager's target.
- *getStatus()* returns a tuple with the ProgressManager's current state, percentage, progress, and target values.
- When an invalid progress or target value is given, the ProgressManager falls into it's 'error state'.
- The functions *isDone()*, *isError()*, *isRunning()* and *isAborted()* return *True* or *False* depending on the ProgressManager's current state.
- External threads can be associated with a ProgressManager using the method *addThreadWatch()*. *closeThreads()* is a blocking function that waits for all associated threads to finish before returning.
- Multiple ProgressManagers can be linked using *addChildProgress()*. Upon getting the parent's state, all child progress and target values will be inherited recursively and summed. The method *isDone()* only returns True if all child processes are done too. Aborting the parent ProgressManager will abort all child ProgressManagers.

- When initializing a ProgressManager with parameter *explicitFinish* = *True*, the 'end state' will not automatically be reached when the progress value hits the target value. Instead, a call to the method *finish()* is needed.

HLAPI functions such as DeviceManager.startLoadInterfaces, *MultiReadWrite.startReadAll()*, *NetworkScanner.startScan()*, etc. automatically create a ProgressManager for you to use. Also notice that these function names begin with *start* (eg. *startReadAll* instead of *readAll*), this will make the functions return immediately to allow your program to continue and handle progress using a ProgressManager.

As an example, we'll be combining databus scanning, device identification and mass reading all in one piece of code, asynchronously.

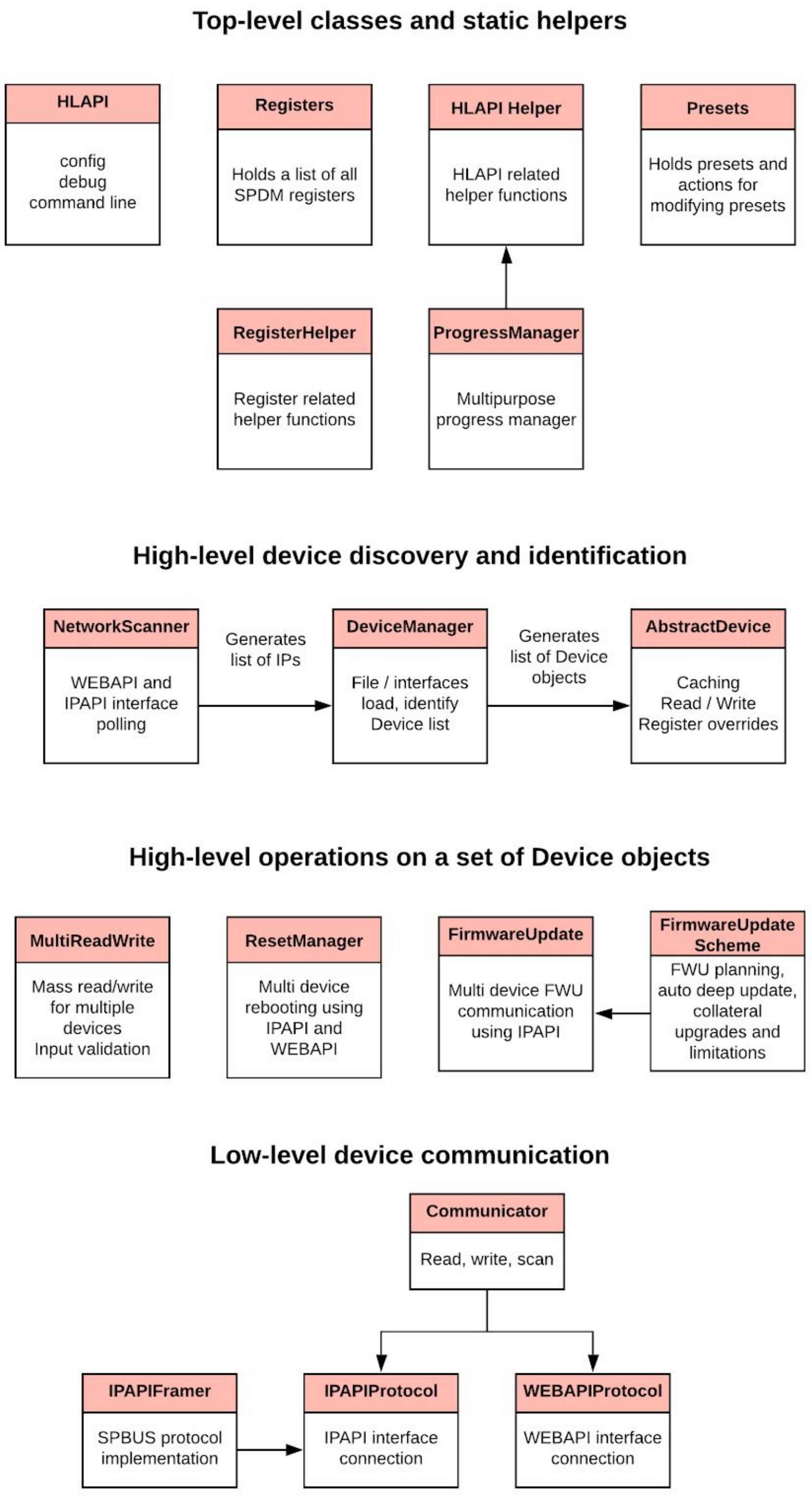
```
import time

from hlapi.hlapi import HLAPI
from hlapi.DeviceManager import DeviceManager
from hlapi.managers.MultiReadWrite import MultiReadWrite

interfaces = {
      "192.168.9.100": {
            "webapi_port": 80,
            "webapi_user": "power",
            "webapi_pass": "power",
            "ipapi_key": "0000000000000000"
      }
}

# read some PDU alert values
mnemonics = ['ssstat', 'ssttri', 'ssitri', 'ssotri', 'ssvtri']

# method to print progress percentage
def showPercentage(progressManager):
      progressManager.waitForInit() # make sure the process has 
started
      last_percentage = 0
      print("0%")
      while progressManager.isRunning():
            percentage = progressManager.getStatus()[1]
            if percentage > last_percentage+10:
                  print(str(percentage)+"%")
                  last_percentage = percentage
            time.sleep(0.1)
      print("100%")
      progressManager.closeThreads() # make sure the process ends

hlapi = HLAPI(debug=False)

# scan databus, identify devices
deviceManager = DeviceManager(hlapi)
deviceManager.startLoadInterfaces(interfaces)
showPercentage(deviceManager.progress)

devices = deviceManager.devices
print(len(devices), "devices found")

# read registers from found devices
multiReadWrite = MultiReadWrite(hlapi, devices)
multiReadWrite.startReadAll(mnemonics)
showPercentage(multiReadWrite.progress)

# pretty-print result
for uid, output in multiReadWrite.result.items():
      print(uid)
      for mnemonic, value in output['data'].items():
            print("\t", mnemonic, value)
```

As you can see, after starting a process, you can make the program do something else until *progressManager.isRunning()* indicates that the process is finished. Once that happens, It is recommended to verify if the ProgressManager finished normally or with an error using *ns.progress.isAborted() or ns.progress.isError().*

```
$ python3 async_progress_manager_demo.py
0%
...
100%
27 devices found
0%
...
100%
192.168.9.100#13
       ssstat 0
       ssvtri 0
       ssotri 0
       ssttri 0
       ssitri 0
192.168.9.100#17
```

```
ssstat 0
ssvtri 0
```

5. HLAPI Schematic


Top-level classes and static helpers
HLAPI
config
debug
command line
Registers
Holds a list of all SPDM registers
HLAPI Helper
HLAPI related helper functions
Presets
Holds presets and actions for modifying presets
RegisterHelper
Register related helper functions
ProgressManager
Multipurpose progress manager
High-level device discovery and identification
NetworkScanner
WEBAPI and IPAPI interface polling
Generates list of IPs
DeviceManager
File / interfaces load, identify Device list
Generates list of Device objects
AbstractDevice
Caching
Read / Write
Register overrides
High-level operations on a set of Device objects
MultiReadWrite
Mass read/write for multiple devices
Input validation
ResetManager
Multi device rebooting using IPAPI and WEBAPI
FirmwareUpdate
Multi device FWU communication using IPAPI
FirmwareUpdate Scheme
FWU planning, auto deep update, collateral upgrades and limitations
Low-level device communication
Communicator
Read, write, scan
IPAPIFramer
SPBUS protocol implementation
IPAPIProtocol
IPAPI interface connection
WEBAPIProtocol
WEBAPI interface connection