g.tec – medical engineering GmbH
Sierningstrasse 14, 4521 Schiedlberg, Austria
Tel.: +43 (7251) 22240-0
Fax: +43 (7251) 22240-39
office@gtec.at, http://www.gtec.at

g.NEEDaccess

NETWORK ENABLED EASY DATA ACCESS INTERFACE

Python Client API 1.18.00
USER MANUAL

To the Reader

Welcome to the medical and electrical engineering world of g.tec!
Discover the only professional biomedical signal processing platform under MATLAB and Simulink. You can develop a wide range of applications with g.tec's hardware, software and complete systems. Our flexible approach lets you choose and combine different components for biosignal recording, amplification, signal processing, stimulation and feedback – all in real-time!

Our team is prepared to find the best solution for your needs.
Take advantage of our experience!

Dr. Christoph Guger Dr. Guenter Edlinger

Researcher and Developer
Reduce development time for sophisticated real-time applications from months to hours.
Integrate g.tec's open platform seamlessly into your processing system.
g.tec's rapid prototyping environment makes creating new applications fun!

Scientist
Explore and even launch new research fields studying real-time interaction with the brain and body.
Process your EEG/ECG/EMG/EOG data with g.tec's biosignal analyzing tools.
Concentrate on your core problems when relying on g.tec's new software features like ICA, AAR or online Hjorth's source derivation.

Study design and data analysis
Are you planning an experiment in the field of brain or life sciences? We can offer consultation in experimental planning, hardware and software selection and can even collect the data for you. If you have already collected EEG/ECG/EMG/EOG, g.tec can perform all of the necessary data analysis, including artifact control, feature extraction, statistical analysis and preparing the results for publication.

How to contact g.tec

☎	+43-7251-22240	**Phone**
	+43-7251-22240-39	**Fax**
	g.tec medical engineering GmbH, Sierningstrasse 14, A-4521 Schiedlberg, Austria	**Mail**
	http://www.gtec.at	**Web**
@	office@gtec.at	**Email**

Table of Contents

1 Safety Notice

In order to use this product safely and fully understand all its functions, make sure to read this manual before using the product.

Medical client applications that incorporate the g.NEEDaccess Server or Client API must be developed according to national/international laws for medical device and software development and must be thoroughly tested before they are used with patients.

Follow the instructions for use for the used PC and the connected devices for allowed environmental conditions.

The used PC must not go to sleep, hibernate, turn off, or launch the screensaver during a measurement.

2 Introduction

g.NEEDaccess is a server service that facilitates simple and platform independent data acquisition from (multiple) devices over a network, and thereby eases the user's workload considerably.

g.NEEDaccess allows users to acquire data easily from g.tec devices without having to take care of low-level data acquisition issues. The server handles data acquisition and preprocessing, so the user receives data ready to analyze.

Since data acquisition is realized over the network, it is now possible to collect the acquired data on a different computer than the g.tec device is connected to (if both are connected to the network). Moreover, the server can provide data from a single acquisition simultaneously to multiple clients. Thus, more than one user can monitor a certain experiment at the same time.

The server software runs as a service in the background of the computer on which it is installed. We informally refer the PC on which this software is running as the server. Biosignal amplifiers are physically connected to the server. Client software is used to interact with the server via the network.

The reference implementation of the server's network API provides a wide range of functions that ease data acquisition and support device-specific operations. The Client API is delivered as a library designed for easy integration into your own projects. It is based on the Client C API, which implements the underlying functionality. These libraries are distributed with the g.NEEDaccess Client API installation package.

This document contains a listing of the methods supported by the .NET Client API only.

For further information on terminology, important concepts, and general insight on how to use the server, as well as an installation guide and system requirements, see the g.NEEDaccess Client API documentation.

3 Intended Use

g.NEEDaccess is intended to be used to acquire and transmit measured biosignal data from g.USBamp, g.HIamp, g.Nautilus PRO and g.Nautilus devices to an application. These biosignals may include electroencephalogram (EEG), electromyogram (EMG), electrooculogram (EOG), and electrocardiogram (ECG).

3.1 Intended use with g.USBamp

Measuring, recording and analysis of electrical activity of the brain (EEG) and/or through the attachment of multiple electrodes at various locations to aid in monitoring and diagnosis as routinely found in clinical settings for the EEG.

3.2 Intended use with g.HIamp

The g.HIamp amplifier is intended to be used to acquire biopotentials and transmit them to a computer via the USB port connection. These biopotentials include the electroencephalogram (EEG), electromyogram (EMG), electrooculogram (EOG), and electrocardiogram (ECG).

3.3 Intended use with g.Nautilus PRO

The g.Nautilus PRO is intended to be used to acquire the electroencephalogram (EEG) and transmit it wirelessly to a computer.

3.4 Intended use with g.Nautilus

The g.Nautilus amplifier is intended to be used to acquire the electroencephalogram (EEG) and/or electrooculogram (EOG) and transmit them wirelessly to a computer.

3.5 Limitation

The device **must not** be used for patient monitoring. The device **must not** be used for the determination of brain death. Additional examinations are needed for diagnosis, and no diagnosis may be done only based on using this device.

4 Introduction to Python Client API

g.NEEDaccess is a g.tec Software tool that provides a Windows service named *GDS* that facilitates simple data acquisition from multiple g.tec devices, also over the local network.

pygds opens this service to the Python world. Its *pygds.GDS* class wraps the C API of g.tec's g.NEEDaccess.

Getting data from a device is as easy as this:

```
import pygds as g
d = g.GDS()
minf_s = sorted(d.GetSupportedSamplingRates()[0].items())[0]
d.SamplingRate, d.NumberOfScans = minf_s
for ch in d.Channels:
    ch.Acquire = True
d.SetConfiguration()
data = d.GetData(d.SamplingRate)
```

You can enter this code interactively in IPython.

With *pygds.Scope*, you can view the data live:

```
scope = g.Scope(1/d.SamplingRate)
d.GetData(d.SamplingRate,scope)
```

Many *pygds.GDS* functions just forward the according g.NEEDaccess functions. Please consult the g.NEEDaccess documentation for the details on these functions.

The functions dealing with callbacks from g.NEEDaccess are not wrapped. They are accessible via *pygds._ffi_dll*, though.

5 Hardware and Software Requirements

Hardware Requirements

The requirements are those of g.NEEDaccess: A PC compatible desktop, notebook workstation running Microsoft Windows. The table below lists optimal settings:

Hardware	Properties
CPU	1.5 GHz or faster processor
Harddisk	32 GB
RAM	4 GB
USB	2.0 high speed port One free USB port (EHCI) for any g.tec device

Software Requirements

pygds requires the installation of g.NEEDaccess server and client API from g.tec.

Software	Version
g.NEEDaccess server and client API	1.18.00
Windows	Windows 10 Pro, English, 64-bit
Acrobat Reader	DC 2018
Python	>= 3.6
numpy	>= 1.13.3
matplotlib	>= 2.1.0
cffi	>= 1.11.2
pywin32	>= 223

6 Installation

Install g.NEEDaccess server and client before installing *pygds*.

pygds uses CFFI. From the C definitions of a C header file, CFFI can learn how to access the functions in a library, specifically in this case the g.NEEDaccess DLL on Windows. Therefore, *pygds* needs the g.NEEDaccess header files, which are part of the g.NEEDaccess Client API, installed in:

```
C:\Program Files\gtec\gNEEDaccess Client API\C\
```

pygds is distributed as whl file.

pygds is installed using pip:

```
pip install pygds.<version>.whl
```

pip will automatically install the required packages CFFI, numpy and matplotlib.

The installed *pygds* consists of only one file:

```
pygds.py
```

It should be available in *C:\Python36\Lib\site-packages* and *C:\Python36\Scripts* (or accordingly in *C:\Python27\..*).

7 Command Line Demos

pygds.py is also a script that can be run from the command line. The script executes demos and tests. Look into *pygds.py* to learn from the demos.

```
usage: pygds.py [-h]

                [--demo [{demo_all,demo_all_api,demo_counter,demo_di
,demo_filter,demo_impedance,demo_remote,demo_save,demo_scaling,demo_
scope,demo_scope_all,demo_usbamp_sync}]]

                [--doctest]



pygds.py runs all demo scripts by default.



optional arguments:

  -h, --help            show this help message and exit

  --demo [{demo_all,demo_all_api,demo_counter,demo_di,demo_filter,de
mo_impedance,demo_remote,demo_save,demo_scaling,demo_scope,demo_scop
e_all,demo_usbamp_sync}]

                        Runs demos. Default is demo_all

  --doctest             Runs doctests
```

8 Usage

The *demo_()* functions in *pygds.py* are an example of how to use *pygds.GDS*.

Basic usage:

```
>>> import pygds
>>> d = pygds.GDS()
```

pygds.GDS hides the API differences between *g.USBamp*, *g.HIamp* and *g.Nautilus*. E.g. *d.GetImpedance()* calls the right function of:

```
GDS_GUSBAMP_GetImpedance()
GDS_GHIAMP_GetImpedance()
GDS_GNAUTILUS_GetImpedance()
```

Similarly, the configuration names are unified. E.g. Trigger means TriggerEnabled, TriggerLinesEnabled or DigitalIOs. See *name_maps*. The device-specific names also work:

```
>>> d.TriggerEnabled == d.Trigger
True
```

For one device, the configuration fields are members of the device object:

```
>>> d.Trigger = True
>>> d.SetConfiguration()
```

For more devices, use the *Configs* list:

```
>>> for c in d.Configs:
...     c.Trigger = True
>>> d.SetConfiguration()
```

pygds.configure_demo() configures all available channels:

```
>>> pygds.configure_demo(d,testsignal=1)
>>> d.SetConfiguration()
```

To acquire a fixed number of samples, please use:

```
>>> a = d.GetData(d.SamplingRate)
>>> a.shape[0] == d.SamplingRate
True
```

To acquire a dynamic number of samples, provide a function *more(samples)*.

A *pygds.Scope* object can be used as *more* parameter of *GetData()*. When closing the scope Window acquisition stops.

```
>>> scope = pygds.Scope(1/d.SamplingRate, title="Channels: %s", ylab
el = u"U[μV]")
>>> a = d.GetData(d.SamplingRate//2,scope)
>>> del scope
>>> a.shape[1]>=d.N_electrodes
True
```

Don't forget *del scope* before repeating this.

To remove a GDS object manually, do:

```
>>> d.Close()
>>> del d
```

In the doctest samples, this is done to make the next test succeed. For a session where only one GDS object is used, there is no need to do this.

9 Reference

9.1 pygds global

9.1.1 pygds.gNEEDaccessHeaders

```
gNEEDaccessHeaders = [
    r"C:/Program Files/gtec/gNEEDaccess Client API/C/GDSClientAPI.h"
,
    r"C:/Program Files/gtec/gNEEDaccess Client API/C/GDSClientAPI_gH
Iamp.h",
    r"C:/Program Files/gtec/gNEEDaccess Client API/C/GDSClientAPI_gN
autilus.h",
    r"C:/Program Files/gtec/gNEEDaccess Client API/C/GDSClientAPI_gU
SBamp.h"]
```

pygds needs these header files and the DLL.

If they are in a different location, you must call pygds.Initialize() manually and provide the right paths.

9.1.2 pygds.GDSError

```
class GDSError(Exception):
```

This is the exception that is raised in case of a g.NEEDaccess API error.

9.1.3 pygds.OpenDevices

```
OpenDevices = None
```

pygds.OpenDevices contains all objects of *pygds.GDS()*. It is used to clean up when exiting python.

9.1.4 pygds.Initialize

```
def Initialize(
    gds_headers=gNEEDaccessHeaders  # default header files used
    , gds_dll=None
):
```

Initializes pygds. This is done automatically at *import pygds*.

If the GDS service is running, then GDSClientAPI.dll is used, else GDSServer.dll. To manually change, first call Uninitialize(), then e.g. Initialize(gds_dll="GDSServer.dll").

pygds.Initialize()

- populates the pygds namespace with definitions from the GDS headers. The *GDS_* prefix is dropped
- loads the GDS client DLL
- calls *GDS_Initialize()*

If g.NEEDaccess is installed in a non-standard location, then *pygds.Initialize()* will fail. Then, you need to call *pygds.Initialize()* manually and provide the header file paths and the DLL path as parameters.

The return value is True if initialization succeeded.

9.1.5 pygds.Uninitialize

```
def Uninitialize():
```

Clean up is done automatically when exiting Python.

Uninitialize() tries not to block, by taking into account these GDS API behaviors:

- *GDS_Uninitialized()* blocks, if called after calling *GDS_Disconnect()* on all connections.
- On the other hand, to prevent a freeze, one must call *GDS_Uninitialized()*, if no device was ever connected, but *GDS_Initialize()* had been called.

9.1.6 pygds.ConnectedDevices

```
class ConnectedDevices(list):
```

Lists all connected devices in a list of type *[(serial, devicetype, inuse)]*:

```
>>> import pygds
>>> cd = pygds.ConnectedDevices()
```

This is used by the *pygds.GDS* constructor. Use it separately only if you don't want to instantiate a pygds.GDS object, but still want to find out which devices are connected.

9.1.7 pygds.ConnectedDevices.find

```
def find(self,
         wanted_type,  # a DEVICE_TYPE_XXX constant
         exclude_serials=None  # list of serials to exclude
         ):
```

Find a device by type.

```
>>> import pygds
>>> cd = pygds.ConnectedDevices()
>>> hiamp = cd.find(pygds.DEVICE_TYPE_GHIAMP)
>>> hiamp is None or len(hiamp.split('.'))>0
True
```

9.1.8 pygds.name_maps

```
name_maps = {
    'GDS_GUSBAMP_CONFIGURATION':
            {
                "SamplingRate": "SampleRate",
                "Counter": "CounterEnabled",
                "Trigger": "TriggerEnabled",
                "DI": "TriggerEnabled",
            },
        'GDS_GHIAMP_CONFIGURATION':
            {
                "SampleRate": "SamplingRate",
```

```
                "Counter": "CounterEnabled",
                "TriggerEnabled": "TriggerLinesEnabled",
                "Trigger": "TriggerLinesEnabled",
                "DI": "TriggerLinesEnabled",
            },
        'GDS_GNAUTILUS_CONFIGURATION':
            {
                "SampleRate": "SamplingRate",
                "Trigger": "DigitalIOs",
                "TriggerEnabled": "DigitalIOs",
                "DI": "DigitalIOs",
            },
        'GDS_GUSBAMP_CHANNEL_CONFIGURATION':
            {
                "Enabled": "Acquire",
                "ReferenceChannel": "BipolarChannel",
            },
        'GDS_GHIAMP_CHANNEL_CONFIGURATION':
            {
                "Enabled": "Acquire",
                "BipolarChannel": "ReferenceChannel",
            },
        'GDS_GNAUTILUS_CHANNEL_CONFIGURATION':
            {
                "Acquire": "Enabled",
                "ReferenceChannel": "BipolarChannel",
            },
        'GDS_GUSBAMP_SCALING':
            {
                "Factor": "ScalingFactor",
            },
}
```

name_maps provides common names for the device-specific configuration fields in order to facilitate code reuse across devices.

9.2 Scope

9.2.1 pygds.Scope

```
class Scope:
```

Scope makes a live update of a Matplotlib diagram and thus simulates an oscilloscope:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from pygds import Scope
>>> import time
>>> f = 10
>>> scope=Scope(1/f)
>>> t = np.linspace(0,100,100)/f
>>> scope(np.array([np.sin(t+i/2) for i in range(10)]))
```

```
True
>>> time.sleep(0.1)
>>> scope(np.array([np.sin(t+i/3) for i in range(10)]))
True
>>> time.sleep(0.1)
>>> scope(np.array([np.sin(t+i/4) for i in range(10)]))
True
>>> time.sleep(0.1)
>>> scope(np.array([np.sin(t+i/5) for i in range(10)]))
True
>>> del scope
```

Scope can be used as the *more* argument of *GetData()* to have a live view on the data.

To use *Scope* as a regular diagram, set *modal=True*.

The object's *__call__(self,scan)* displays the scans. On the first call to the object (via *__call__()*), the diagram is initialized. The *scans* parameter of *__call__()* is an *(n,ch)* numpy array. It must have the same shape at every call.

9.3 GDS

9.3.1 pygds.GDS

```
class GDS(_config_wrap):
```

The *pygds.GDS* class initializes the connection to g.NEEDaccess server.

The constructor

- initializes the connection to the wanted device(s) and
- fetches the configuration(s).

gds_device: can be

- omitted (default)
- the first letter of the serial
- one of DEVICE_TYPE_GUSBAMP, DEVICE_TYPE_GHIAMP or DEVICE_TYPE_GNAUTILUS
- a single serial
- comma-separated serials

exclude_serials: a list or set of serials to ignore. Default: None.

server_ip: the IP address of the GDS server. Default: pygds.SERVER_IP

- The g.NEEDaccess server port is pygds.SERVER_PORT and it is fixed.
- The client by default is pygds.CLIENT_IP and pygds.CLIENT_PORT. For a remote *server_ip,* the local IP is automatically determined.

Without parameters, the localhost g.NEEDaccess server is used and the first available device is connected.

For one device, the configuration fields are members of the GDS object. For more devices, every configuration is an entry in the *Configs* member.

g.USBamp config:

Name	String holding the serial number of g.HIamp (e.g. *UB-2014.01.02*)
DeviceType	*pygds.DEVICE_TYPE_XXX* constant representing the device type (predefined, must not be changed)
SamplingRate	Specify the sampling frequency of g.HIamp in Hz as unsigned integer
NumberOfScans	Specify the buffering block size as unsigned short, possible values depend on sampling rate, use function *GetSupportedSamplingRates()* to get recommended values.
CommonGround	Array of 4 bool elements to enable or disable common ground
CommonReference	Array of 4 bool values to enable or disable common reference
ShortCutEnabled	Bool enabling or disabling g.USBamp shortcut
CounterEnabled	Show a counter on first recorded channel that is incremented with every block transmitted to the PC. Overruns at 1000000.
TriggerEnabled	Scan the digital trigger channel with the analog inputs
InternalSignalGenerator.Enabled	Apply internal test signal to all inputs
InternalSignalGenerator.WaveShape	Unsigned integer representing the wave shape of the internal test signal. Can be 0=square, 1=saw tooth, 2=sine 3=DRL or 4=noice See g.GUSBAMP_WAVESHAPE_XXX constants.
InternalSignalGenerator.Amplitude	The amplitude of the test signal (can be -250 to 250 mV)
InternalSignalGenerator.Offset	The offset of the test signal (can be -200 to 200 mV)
InternalSignalGenerator.Frequency	The frequency of the test signal (can be 1 to 100 Hz)
Channels	Array of g.USBamp channel configurations (gUSBampChannels) holding properties for each analog channel
Channels[i].ChannelNumber	Unsigned integer holding the channel number of the analog channel
Channels[i].Acquire	Bool value selecting the channel for data acquisition
Channels[i].BandpassFilterIndex	Perform a digital bandpass filtering of the input channels. Use *GetBandpassFilters()* to get filter indices.

Channels[i].NotchFilterIndex	Perform a bandstop filtering to suppress the power line frequency of 50 Hz or 60 Hz. Use *GetNotchFilters()* to get filter indices.
Channels[i].BipolarChannel	Select a channel number as reference channel for an analog channel

g.HIamp config:

Name	String holding the serial number of g.HIamp (e.g. *HA-2014.01.02*)
DeviceType	*pygds.DEVICE_TYPE_XXX* constant representing the device type (predefined, must not be changed)
SamplingRate	Specify the sampling frequency of g.Hiamp in Hz as unsigned integer
NumberOfScans	Specify the buffering block size as unsigned short, possible values depend on sampling rate, use function *GetSupportedSamplingRates()* to get recommended values
CounterEnabled	Show a counter on first recorded channel which is incremented with every block transmitted to the PC. Overruns at 1000000
TriggerLinesEnabled	Scan the digital trigger channel with the analog inputs
HoldEnabled	Enable signal hold
Channels	Array of g.HIamp channel configurations holding properties for each analog channel
Channels[i].ChannelNumber	Unsigned integer holding the channel number of the analog channel
Channels[i].Acquire	Bool value selecting the channel for data acquisition
Channels[i].BandpassFilterIndex	Perform a digital bandpass filtering of the input channels. Use *GetBandpassFilters()* to get filter indices
Channels[i].NotchFilterIndex	Perform a bandstop filtering to suppress the power line frequency of 50 Hz or 60 Hz. Use *GetNotchFilters()* to get filter indices
Channels[i].ReferenceChannel	Select a channel number as reference channel for an analog channel
InternalSignalGenerator.Enabled	Apply internal test signal to all inputs (requires shortcut of all analog channels to ground)
InternalSignalGenerator.Frequency	Specify the frequency of the test signal. Fix: Amplitude = -Offset = 7.62283 mV.

g.Nautilus config:

Name	String holding the serial number of g.Nautilus (e.g. NA-2014.07.67)

DeviceType	*pygds.DEVICE_TYPE_XXX* constant representing the device type
SamplingRate	Specify the sampling frequency of g.Nautilus in Hz as unsigned integer
NumberOfScans	Specify the buffering block size as unsigned short, possible values depend on sampling rate, use function *GetSupportedSamplingRates()* to get recommended values
InputSignal	Holds type of input signal, can be 0=Electrode, 1=Shortcut or 5=TestSignal. See *pygds.GNAUTILUS_INPUT_SIGNAL_XXX* constants.
NoiseReduction	Bool value enabling noise reduction for g.Nautilus
CAR	Bool value enabling common average calculation for g.Nautilus
AccelerationData	Bool value enabling acquisition of acceleration data from g.Nautilus head stage, adds 3 additional channels to the data acquisition for x, y, and z direction
Counter	show a counter as an additional channel
LinkQualityInformation	Bool value enabling additional channel informing about link quality between head stage and base station
BatteryLevel	Bool to enable acquisition of additional channel holding information about remaining battery capacity
DigitalIOs	Scan the digital channels with the analog inputs and add them as additional channel acquired
ValidationIndicator	Enables the additional channel validation indicator, informing about the liability of the data recorded
NetworkChannel	Unsigned integer value representing the network channel used between head stage and base station
Channels	Array of g.Nautilus channel configurations holding properties for each analog channel
Channels[i].ChannelNumber	Unsigned integer holding the channel number of the analog channel
Channels[i].Enabled	Bool value selecting the channel for data acquisition
Channels[i].Sensitivity	Double value representing the sensitivity of the specified channel
Channels[i].UsedForNoiseReduction	Bool value indicating if channel should be used for noise reduction
Channels[i].UsedForCAR	Bool value indicating if channel should be used for common average calculation

Channels[i].BandpassFilterIndex	Perform a digital bandpass filtering of the input channels. Use *GetBandpassFilters()* to get filter indices.
Channels[i].NotchFilterIndex	Perform a bandstop filtering to suppress the power line frequency of 50 Hz or 60 Hz. Use *GetNotchFilters()* to get filter indices.
Channels[i].BipolarChannel	Select a zero based channel index as reference channel for an analog channel

Note that some names are unified to work for all devices. See *name_maps*.

9.3.2 pygds.GDS.SetConfiguration

```
def SetConfiguration(self):
```

SetConfiguration() needs to be called to send the configuration to the device.

Before calling the underlying *GDS_SetConfiguration()*, the channels that are not available on the device are removed. So one can do *for ch in d.Channels: ch.Acquire=True* without the need to consult *GetAvailableChannels()*.

9.3.3 pygds.GDS.GetConfiguration

```
def GetConfiguration(self):
```

GetConfiguration() fetches the configuration from the device. This is done automatically when instantiating a GDS object.

9.3.4 pygds.GDS.GetDataInfo

```
def GetDataInfo(self,
                scanCount  # number of scans
                ):
```

GetDatatInfo() returns (channelsPerDevice, bufferSizeInSamples).

channelsPerDevice is a list of channels for each device.

bufferSizeInSamples is the total number of samples.

```
>>> import pygds
>>> d = pygds.GDS()
>>> scanCount = 500
>>> channelsPerDevice, bufferSizeInSamples = d.GetDataInfo(scanCount
)
>>> sum(channelsPerDevice)*scanCount == bufferSizeInSamples
True
>>> d.Close(); del d
```

9.3.5 pygds.GDS.N_ch_calc

```
def N_ch_calc(self):
```

N_ch_calc() returns the number of configured channels. After the first call, you can use *d.N_ch* to get the number of configured channels.

```
>>> import pygds; d = pygds.GDS()
>>> n = d.N_ch_calc()
>>> d.N_ch == n
True
>>> d.Close(); del d
```

d.N_electrodes is the number of electrodes in the GDS connection for all connected devices. *d.N_ch* can be equal, smaller or larger than *d.N_electrodes*, depending on the configuration.

9.3.6 pygds.GDS.NumberOfScans_calc

```
def NumberOfScans_calc(self):
```

```
sr = self.GetSupportedSamplingRates()
for i, c in enumerate(self.Configs):
    c.NumberOfScans = sr[i].get(c.SamplingRate, 8)
```

Sets *d.NumberOfScans* by mapping *d.SamplingRate* via *GetSupportedSamplingRates()*.

9.3.7 pygds.GDS.IndexAfter

```
def IndexAfter(self,
                chname=''  # '1',..,'Counter',... => index after; ''
=> channel count
                ):
```

Get the channel 0-based index one position after the 1-based *chname*. *chname* can also be one of:

```
Counter
Trigger
```

and for g.Nautilus also:

```
AccelerationData
LinkQualityInformation
BatteryLevel
DigitalIOs
ValidationIndicator
```

Without *chname* it gives the count of configured channels.

For more devices per GDS object one can use:

```
name+serial, e.g. 1UB-2008.07.01
```

to get the index of a channel of a specific device.

```
>>> import pygds; d = pygds.GDS()
>>> d.IndexAfter('4'+d.Name)
4
>>> d.IndexAfter('4')
4
```

```
>>> d.IndexAfter('AccelerationData')>=0
True
>>> d.IndexAfter('Counter')>=0
True
>>> d.IndexAfter('LinkQualityInformation')>=0
True
>>> d.IndexAfter('BatteryLevel')>=0
True
>>> d.IndexAfter('DigitalIOs')>=0
True
>>> d.IndexAfter('Trigger')>=0
True
>>> d.IndexAfter('ValidationIndicator')>=0
True
>>> d.IndexAfter('')==d.N_ch_calc()
True
>>> d.Close(); del d
```

9.3.8 pygds.GDS.GetData

```
def GetData(self,
            # number of scans. A scan is a sample for each channel.
            scanCount,
            more=None # a function that takes the samples and must r
eturn True if more samples are wanted
            ):
```

`GetData()` gets the data from the device.

GetData allocates `scanCount*N_ch*4` memory two times. It fills one copy in a separate thread with sample data from the device, while the other copy is processed by the `more` function in the current thread. Then it swaps the two buffers.

`more(samples)` gets the current samples and decides, whether to continue acquisition by returning True.

`more` must copy the samples to reuse them later.

```
>>> import pygds; d = pygds.GDS()
>>> samples = []
>>> more = lambda s: samples.append(s.copy()) or len(samples)<2
>>> data=d.GetData(d.SamplingRate, more)
>>> len(samples)
2
>>> d.Close(); del d
```

9.3.9 pygds.GDS.GetAvailableChannels

```
def GetAvailableChannels(self,
                         combine=True  # and-combine the C API's GDS
_XXX_GetAvailableChannels with the channel's Acquire flag
                         ):
```

GetAvailableChannels() wraps C API's *GDS_XXX_GetAvailableChannels()*. The return value of each device is an entry in the returned list. *d.GetAvailableChannels()[0]* is a list of 0 or 1.

This is called when instantiating a GDS object to initialize the *N_electrodes* member. It is also called in *SetConfiguration()* to ignore the channels that are not available. And it is called in *IndexAfter()* and thus also in *N_ch_calc()* to get the channel index or the configured channel count. There should be no reason to call this directly.

9.3.10 pygds.GDS.GetAvailableDigitalIOs

```
def GetAvailableDigitalIOs(self):
```

GetAvailableDigitalIOs() wraps the g.Nautilus *GDS_GNAUTILUS_GetAvailableDigitalIOs()*. g.Nautilus only.

The return value of each device is an entry in the returned list. *d.GetAvailableDigitalIOs()[0]* is a list of dicts, each with these keys:

ChannelNumber	Unsigned integer representing the digital IO number
Direction	String representing the direction of the digital channel (In=0 or Out=1)

9.3.11 pygds.GDS.GetAsyncDigitalIOs

```
def GetAsyncDigitalIOs(self):
```

GetAsyncDigitalIOs() wraps the g.USBamp *GDS_GUSBAMP_GetAsyncDigitalIOs()*. g.USBamp only.

The return value of each device is an entry in the returned list. *d.GetAsyncDigitalIOs()[0]* is a list of dicts, each with these keys:

ChannelNumber	Integer value representing the digital channel number
Direction	String holding the digital channel direction (In=0 or Out=1)
Value	Current value of the digital channel (true or false)

9.3.12 pygds.GDS.SetAsyncDigitalOutputs

```
def SetAsyncDigitalOutputs(self, outputs):
```

SetAsyncDigitalOutputs() wraps the g.USBamp *GDS_GUSBAMP_SetAsyncDigitalOutputs()*. g.USBamp only.

9.3.13 pygds.GDS.GetDeviceInformation

```
def GetDeviceInformation(self):
```

GetDeviceInformation() wraps the C API's *GDS_XXX_GetDeviceInformation()* functions.

The device information for each device is a string entry in the returned list.

9.3.14 pygds.GDS.GetImpedance

```
def GetImpedance(self,
                  # list of bool or int telling which electrode is ac
tive (g.HIamp only)
                  active=None
                  ):
```

`GetImpedance()` wraps the C API's `GDS_XXX_GetImpedance()` functions.

Gets the impedances for all channels of all devices. The impedances of each device are a list entry in the returned list.

Note, that for g.Nautilus electrode 15 = Cz must be connected to GND, else an exception occurs.

```
>>> import pygds; d = pygds.GDS()
>>> imps = d.GetImpedance([1]*len(d.Channels))
>>> len(imps[0])==len(d.Channels)
True
>>> d.Close(); del d
```

9.3.15 pygds.GDS.GetScaling

```
def GetScaling(self):
```

`GetScaling()` wraps the C API's `GDS_XXX_GetScaling()` functions.

The return value of each device is a dict entry in the returned list. Each dict has the fields:

Factor	Array holding single type values with scaling factor for each analog channel.
Offset	Array holding single type values with offset for each analog channel.

9.3.16 pygds.GDS.Calibrate

```
def Calibrate(self):
```

`Calibrate()` wraps the C API's `GDS_XXX_Calibrate()` functions(), which calibrates the device.

The return value of each device is a dict entry in the returned list. d.Calibrate()[0] is a dict with these keys:

ScalingFactor	Array holding single type values with scaling factor for each analog channel.
Offset	Array holding single type values with offset for each analog channel.

9.3.17 pygds.GDS.SetScaling

```
def SetScaling(self,
               scaling  # an array of GDS_XXX_SCALING structs or equ
ivalent dicts
               ):
```

`SetScaling()` wraps the C API's `GDS_XXX_SetScaling()` functions.

`SetScaling()` sets the scaling on the device.

9.3.18 pygds.GDS.ResetScaling

```
def ResetScaling(self):
```

ResetScaling() wraps the g.Nautilus *GDS_GNAUTILUS_ResetScaling()* function.

The scaling is reset to Offset=0.0 and Factor=1.0. g.Nautilus only.

9.3.19 pygds.GDS.GetNetworkChannel

```
def GetNetworkChannel(self):
```

GetNetworkChannel() wraps the C API's *GDS_GNAUTILUS_GetNetworkChannel()*.

The currently used g.Nautilus network channel is an entry in the returned list.

9.3.20 pygds.GDS.GetFactoryScaling

```
def GetFactoryScaling(self):
```

GetFactoryScaling() wraps C API's *GDS_GHIAMP_GetFactoryScaling()*.

The factory scaling is an entry for each g.HIamp in the returned list. Only g.HIamp.

9.3.21 pygds.GDS.GetSupportedSamplingRates

```
def GetSupportedSamplingRates(self):
```

GetSupportedSamplingRates() wraps the C API's *GDS_XXX_GetSupportedSamplingRates()* functions.

For each device a dict *{SamplingRate:NumberOfScans}* is an entry in the returned list.

You can do *d.NumberOfScans=d.GetSupportedSamplingRates()[0][d.SamplingRate]* to set the recommended NumberOfScans. This is done when using *d.NumberOfScans_calc()*, and if there are more devices per GDS object.

9.3.22 pygds.GDS.GetBandpassFilters

```
def GetBandpassFilters(self):
```

GetBandpassFilters() wraps the C API's *GDS_XXX_GetBandpassFilters()* functions.

In the returned list, an entry per device is a list of dicts, with one dict for each filter. The dicts also contain the key *BandpassFilterIndex* to be used to set the filter.

The fields per filter are:

BandpassFilter Index	Use this for the according channel field
SamplingRate	Double value holding the sampling rate for which the filter is valid
Order	Unsigned integer holding filter order
LowerCutoffFrequency	Double representing lower cutoff frequency of the filter

UpperCutoffFrequency	Double representing upper cutoff frequency of the filter
TypeId	Representing type of filter

To choose a filter for the desired sampling rate, you can do this:

```
>>> import pygds; d = pygds.GDS()
>>> f_s_2 = sorted(d.GetSupportedSamplingRates()[0].items())[1] #512
or 500
>>> d.SamplingRate, d.NumberOfScans = f_s_2
>>> BP = [x for x in d.GetBandpassFilters()[0] if x['SamplingRate']
== d.SamplingRate]
>>> for ch in d.Channels:
...     ch.Acquire = True
...     if BP:
...         ch.BandpassFilterIndex = BP[0]['BandpassFilterIndex']
>>> d.SetConfiguration()
>>> d.GetData(d.SamplingRate).shape[0] == d.SamplingRate
True
>>> d.Close(); del d
```

9.3.23 pygds.GDS.GetNotchFilters

```
def GetNotchFilters(self):
```

GetNotchFilters() wraps the C API's *GDS_XXX_GetNotchFilters()* functions.

In the returned list, an entry per device is a list of dicts, with one dict for each filter. The dicts also contain the key *NotchFilterIndex* to be used to set the filter.

The fields per filter are:

NotchFilterIndex	Use this for the according channel field
SamplingRate	Double value holding the sampling rate for which the filter is valid
Order	Unsigned integer holding filter order
LowerCutoffFrequency	Double representing lower cutoff frequency of the filter
UpperCutoffFrequency	Double representing upper cutoff frequency of the filter
TypeId	Representing type of filter

To choose a filter for the desired sampling rate, you can do this:

```
>>> import pygds; d = pygds.GDS()
>>> f_s_2 = sorted(d.GetSupportedSamplingRates()[0].items())[1] #512
or 500
>>> d.SamplingRate, d.NumberOfScans = f_s_2
>>> N = [x for x in d.GetNotchFilters()[0] if x['SamplingRate'] == d
.SamplingRate]
>>> for ch in d.Channels:
...     ch.Acquire = True
```

```
...     if N:
...         ch.NotchFilterIndex = N[0]['NotchFilterIndex']
>>> d.SetConfiguration()
>>> d.GetData(d.SamplingRate).shape[0] == d.SamplingRate
True
>>> d.Close(); del d
```

9.3.24 pygds.GDS.GetSupportedSensitivities

```
def GetSupportedSensitivities(self):
```

GetSupportedSensitivities() wraps the C API's *GDS_GNAUTILUS_GetSupportedSensitivities()*.

The supported sensitivities for each g.Nautilus device are an entry in the returned list. g.Nautilus only.

d.GetSupportedSensitivities()[0] is a list of integers. Each integer can be used as the channel's Sensitivity.

9.3.25 pygds.GDS.GetSupportedNetworkChannels

```
def GetSupportedNetworkChannels(self):
```

GetSupportedNetworkChannels() wraps C API's *GDS_GNAUTILUS_GetSupportedNetworkChannels()*.

The supported network channels for each g.Nautilus device are an entry in the returned list. g.Nautilus only.

GetSupportedNetworkChannels()[0] is a list of integers. Each integer can be used in *d.SetNetworkChannel()*.

9.3.26 pygds.GDS.GetSupportedInputSources

```
def GetSupportedInputSources(self):
```

GetSupportedInputSources() function wraps *GDS_GNAUTILUS_GetSupportedInputSources()*.

The supported g.Nautilus input sources for each g.Nautilus device are an entry in the returned list. g.Nautilus only.

d.GetSupportedInputSources()[0] is a list of integers corresponding to the *pygds.GDS_GNAUTILUS_INPUT_XXX* constants. Each integer can be used for *d.InputSignal*.

9.3.27 pygds.GDS.GetChannelNames

```
def GetChannelNames(self):
```

GetChannelNames() wraps C API's *GDS_GNAUTILUS_GetChannelNames()*.

A list of channel names for each g.Nautilus device is an entry in the returned list. g.Nautilus only.

d.GetChannelNames()[0] is a list of strings. The strings correspond to the labels on the electrodes.

9.3.28 pygds.GDS.SetNetworkChannel

```
def SetNetworkChannel(self,
                      networkchannels  # integer, or list of integer
s in case of more attached devices
                      ):
```

SetNetworkChannel() wraps the C API's *GDS_GNAUTILUS_SetNetworkChannel()*. g.Nautilus only.

SetNetworkChannel() sets the g.Nautilus network channel.

networkchannels is one of the integers returned by GetSupportedNetworkChannels().

9.3.29 pygds.GDS.Close

```
def Close(self):
```

Closes the device.

All GDS objects are removed automatically when exiting Python.

To remove a GDS object manually, use:

```
d.Close()
del d
```

10 Demo Code

10.1 pygds.configure_demo

```
def configure_demo(d, testsignal=False, acquire=1):

if d.DeviceType == DEVICE_TYPE_GNAUTILUS:
    sensitivities = d.GetSupportedSensitivities()[0]
    d.SamplingRate = 250
    if testsignal:
        d.InputSignal = GNAUTILUS_INPUT_SIGNAL_TEST_SIGNAL
    else:
        d.InputSignal = GNAUTILUS_INPUT_SIGNAL_ELECTRODE
else:
    d.SamplingRate = 256
    d.InternalSignalGenerator.Enabled = testsignal
    d.InternalSignalGenerator.Frequency = 10
d.NumberOfScans_calc()
d.Counter = 0
d.Trigger = 0
for i, ch in enumerate(d.Channels):
    ch.Acquire = acquire
    ch.BandpassFilterIndex = -1
    ch.NotchFilterIndex = -1
    ch.BipolarChannel = 0  # 0 => to GND
    if d.DeviceType == DEVICE_TYPE_GNAUTILUS:
        ch.BipolarChannel = -1  # -1 => to GND
        ch.Sensitivity = sensitivities[5]
        ch.UsedForNoiseReduction = 0
        ch.UsedForCAR = 0
#not unified
if d.DeviceType == DEVICE_TYPE_GUSBAMP:
    d.ShortCutEnabled = 0
    d.CommonGround = [1]*4
    d.CommonReference = [1]*4
    d.InternalSignalGenerator.WaveShape = GUSBAMP_WAVESHAPE_SINE
    d.InternalSignalGenerator.Amplitude = 200
    d.InternalSignalGenerator.Offset = 0
elif d.DeviceType == DEVICE_TYPE_GHIAMP:
    d.HoldEnabled = 0
elif d.DeviceType == DEVICE_TYPE_GNAUTILUS:
    d.NoiseReduction = 0
    d.CAR = 0
    d.ValidationIndicator = 1
    d.AccelerationData = 1
    d.LinkQualityInformation = 1
    d.BatteryLevel = 1
```

Makes a configuration for the demos.

The device configuration fields are members of the device object d. If d.ConfigCount>1, i.e. more devices are connected, use d.Configs[i] instead.

Config names are unified: See name_maps.

This does not configure a filter. Note that g.HIamp version < 1.0.9 will have wrong first value without filters.

10.2 pygds.demo_counter

```
def demo_counter():

d = GDS()
# configure
configure_demo(d, testsignal=d.DeviceType != DEVICE_TYPE_GUSBAMP)
d.Counter = 1
# set configuration
d.SetConfiguration()
# get data
data = d.GetData(d.SamplingRate)
# plot counter
scope = Scope(1/d.SamplingRate, modal=True, ylabel='n',
              xlabel='t/s', title='Counter')
icounter = d.IndexAfter('Counter')-1
scope(data[:, icounter:icounter+1])
plt.show()
# plot second channel
scope = Scope(1/d.SamplingRate, modal=True, ylabel=u'U/μV',
              xlabel='t/s', title='Channel 2')
scope(data[:, 1:2])
# or
# plt.plot(data[1:,1])
#plt.title('Channel 2')
plt.show()
# close
d.Close()
del d
```

This demo

- configures to internal test signal
- records 1 second
- displays the counter
- displays channel 2

Have a device

- connected to the PC and
- switched on

10.3 pygds.demo_save

```
def demo_save():

filename = 'demo_save.npy'
assert not os.path.exists(
    filename), "the file %s must not exist yet" % filename
```

```python
# device object
d = GDS()
# configure
configure_demo(d, testsignal=True)
# set configuration
d.SetConfiguration()
# get data
data = d.GetData(d.SamplingRate)
# save
np.save(filename, data)
del data
# load
dfromfile = np.load(filename)
os.remove(filename)
# show loaded
scope = Scope(1/d.SamplingRate, modal=True,
              xlabel="t/s", title='Channel 1')
scope(dfromfile[:, 0:1])
plt.show()
# close
d.Close()
del d
```

This demo

- records the internal test signal
- saves the acquired data after recording

Have a device

- connected to the PC and
- switched on

10.4 pygds.demo_di

```python
def demo_di():

d = GDS()
# configure
configure_demo(d, testsignal=True, acquire=0)
d.Trigger = 1
d.Channels[0].Acquire = 1  # at least one channel needs to be there
d.SetConfiguration()
# initialize scope object
scope = Scope(1/d.SamplingRate, subplots={0: 0, 1: 1}, xlabel=(
    '', 't/s'), ylabel=(u'V/μV', 'DI'), title=('Ch1', 'DI'))
# get data to scope
d.GetData(d.SamplingRate, more=scope)
di1 = d.IndexAfter('DI')-1
di2 = d.IndexAfter('Trigger')-1
assert di1 == di2
print('DI channel is ', di1)
```

```
# close
d.Close()
del d
```

This demo

- records the DI channel
- displays it with the live scope

Have a device

- connected to the PC and
- switched on

10.5 pygds.demo_scope

```
def demo_scope():

d = GDS()
# configure
configure_demo(d, testsignal=True, acquire=0)
d.Channels[0].Acquire = 1  # at least one channel needs to be there
d.SetConfiguration()
# initialize scope
scope = Scope(1/d.SamplingRate, xlabel='t/s', ylabel=u'V/μV',
              title="Internal Signal Channels: %s")
# get data to scope
d.GetData(d.SamplingRate, more=scope)
# close
d.Close()
del d
```

This demo

- records a test signal
- displays it in the live scope

Have a device

- connected to the PC and
- switched on

10.6 pygds.demo_scope_all

```
def demo_scope_all():

d = GDS()
# configure
configure_demo(d, testsignal=True, acquire=1)
sr = d.GetSupportedSamplingRates()[0]
d.SamplingRate = max(sr.keys())
if d.DeviceType == DEVICE_TYPE_GHIAMP:
    for i, ch in enumerate(d.Channels):
        if i >= 40:
            ch.Acquire = 0
```

```
elif d.DeviceType == DEVICE_TYPE_GUSBAMP:
    d.SamplingRate = 1200  # >1200 no internal signal
d.NumberOfScans = sr[d.SamplingRate]
d.SetConfiguration()
# initialize scope
scope = Scope(1/d.SamplingRate, xlabel='t/s', ylabel=u'V/μV',
              title="Internal Signal Channels: %s")
# get data every 1/3 of a second
cnt = d.SamplingRate//3  # determines how often an update happens
d.GetData(cnt, more=scope)
# close
d.Close()
del d
```

This demo

- records a test signal for all channels with maximum sampling rate
- displays it in the live scope

Have a device

- connected to the PC and
- switched on

10.7 pygds.demo_scaling

```
def demo_scaling():

d = GDS()
# get scaling
current_scaling = d.GetScaling()
print(current_scaling)
# close
d.Close()
del d
```

This demo tests the function GetScaling.

Have a device

- connected to the PC and
- switched on

10.8 pygds.demo_impedance

```
def demo_impedance():

d = GDS()
# get impedances
impedances = d.GetImpedance()
print(impedances[0])
# close
d.Close()
del d
```

This demo demonstrates impedance measurement.

Have a device

- connected to the PC and
- switched on
- for g.Nautilus, Cz must be connected to GND

10.9 pygds.demo_filter

```
def demo_filter():

d = GDS()
# configure to the second lowest sampling rate
f_s_2 = sorted(d.GetSupportedSamplingRates()[0].items())[1]
d.SamplingRate, d.NumberOfScans = f_s_2
# get all applicable filters
N = [x for x in d.GetNotchFilters()[0] if x['SamplingRate']
     == d.SamplingRate]
BP = [x for x in d.GetBandpassFilters()[0] if x['SamplingRate']
      == d.SamplingRate]
# set the first applicable filter
for ch in d.Channels:
    ch.Acquire = True
    if N:
        ch.NotchFilterIndex = N[0]['NotchFilterIndex']
    if BP:
        ch.BandpassFilterIndex = BP[0]['BandpassFilterIndex']
# set configuration on device
d.SetConfiguration()
# get and display one second of data
Scope(1/d.SamplingRate, modal=True)(d.GetData(d.SamplingRate))
plt.show()
# You wouldn't do the following. Here it is just to check GetConfigu
ration() functionality.
for ch in d.Channels:
    ch.Acquire = False
    ch.NotchFilterIndex = -1
    ch.BandpassFilterIndex = -1
d.GetConfiguration()
assert d.Channels[0].Acquire == True
assert d.Channels[0].NotchFilterIndex != - \
    1 or d.Channels[0].BandpassFilterIndex != -1
```

This demo demonstrates the use of filters.

Have a device

- connected to the PC and
- switched on

10.10 pygds.demo_all_api

```
def demo_all_api():

print("Testing communication with the devices")
print("======================================")
print()
# device object
d = GDS()
# configure
configure_demo(d)
d.Counter = True
d.SetConfiguration()
# print all Configs
print("Devices:")
for c in d.Configs:
    print(str(c))
    print()
print()
# calc number of channels
print("Configured number of channels: ", d.N_ch_calc())
print()
# available channels
print("Available Channels: ", d.GetAvailableChannels())
print()
# device info string
print("Device informations:")
dis = d.GetDeviceInformation()
for di in dis:
    print(di)
    print()
print()
# supported sampling rates
print("Supported sampling rates: ")
for sr in d.GetSupportedSamplingRates():
    for x in sr:
        print(str(x))
print()
# impedances
print("Measure impedances: ")
try:
    imps = d.GetImpedance()
    print(imps)
except GDSError as e:
    print(e)
print()
# filters
print("Bandpass filters:")
bps = d.GetBandpassFilters()
for bp in bps:
    for abp in bp:
```

```
        print(str(abp))
print()
print("Notch filters:")
notchs = d.GetNotchFilters()
for notch in notchs:
    for anotch in notch:
        print(str(anotch))
print()
# device specific functions
```

This demo calls all wrapped API functions. It can be used as a regression test.

Have a device

- connected to the PC and
- switched on

10.11 pygds.demo_usbamp_sync

```
def demo_usbamp_sync():

dev_names = [n for n, t, u in ConnectedDevices() if t ==
             DEVICE_TYPE_GUSBAMP and not u]
devices = ','.join(dev_names)
print('master,slave = ', devices)
print()
if len(dev_names) == 2:
    d = GDS(devices)
    # configure each
    for c in d.Configs:
        c.SamplingRate = 256
        c.NumberOfScans = 8
        c.CommonGround = [0]*4
        c.CommonReference = [0]*4
        c.ShortCutEnabled = 0
        c.CounterEnabled = 0
        c.TriggerEnabled = 0
        c.InternalSignalGenerator.Enabled = 1
        c.InternalSignalGenerator.Frequency = 10
        c.InternalSignalGenerator.WaveShape = GUSBAMP_WAVESHAPE_SINE
        c.InternalSignalGenerator.Amplitude = 200
        c.InternalSignalGenerator.Offset = 0
        for ch in c.Channels:
            ch.Acquire = 1
            # do not use filters
            ch.BandpassFilterIndex = -1
            ch.NotchFilterIndex = -1
            # do not use a bipolar channel
            ch.BipolarChannel = 0
    d.SetConfiguration()
    # create time scope
    scope = Scope(1/c.SamplingRate, xlabel='t/s',
```

```
                  ylabel=u'V/µV', title="%s Channels")
    # make scope see 1 second
    d.GetData(1*c.SamplingRate, more=scope)
    # close
    d.Close()
    del d
```

This demo

- configures two g.USBamp with the sinus test signal
- records all 32 channels of the two synchronized g.USBamp and
- displays all 32 channels in the time scope.

Have two switched on g.USBamp devices

- connected to the PC and
- connected with each other via the synch cables

10.12 pygds.demo_remote

```
def demo_remote():

ip = _this_ip('10.255.255.255')  # use this IP for the test
print("connecting "+ip)
for ch in 'U H N'.split():
    try:
        # provide server_ip when connecting remote PC
        d = GDS(gds_device=ch, server_ip=ip)
        print(d.GetDeviceInformation()[0])
        d.Close()
        del d
    except:
        pass
```

This demo shows how to connect a remote PC.

Have a device

- connected to the PC and
- switched on

10.13 pygds.demo_all

```
def demo_all():
```

Runs all demos.

GUGER
TECHNOLOGIES

contact information

g.tec medical engineering GmbH
Sierningstrasse 14
4521 Schiedlberg
Austria

tel. +43 7251 22240
fax. +43 7251 22240 39
web: www.gtec.at
e-mail: office@gtec.at