# Python Programming Manual

## 1. Introduction

The royalty free Python drivers represent a proprietary interface specifically developed for silicon systems devices. This document provides an explanation of the functions and methods available to application developers. Any software code examples given in this document are for information only and free to use in custom applications.
Python 3.2 and above (32-Bit and 64-Bit) for Microsoft® Windows™, Linux®, Raspberry Pi and Android™ is supported. It is advisable to always utilize the most recent version of the driver for the best compatibility and performance. All drivers can be downloaded from http://siliconsystems.at.
If the DHCP-assigned IP address of a device is not known, the royalty free silicon systems Device Manager can be used to find all devices in the local area network. In addition, the software shows various properties, the health status and calibration validity of each device. It can be downloaded from http://siliconsystems.at/support.php?download.

## 2. Common Device Functionality

The functions provided by the silicon systems driver package can be divided into two subsets: device-specific functions (e.g. reading the temperature of temperature monitor device) and common functions (e.g. retrieving the status of a device or finding available devices in the local area network). In this section the latter functions are explained.

### 2.1. Loading the Driver

Before any methods provided by the driver class can be utilized, the package needs to be imported. The appropriate dynamic link library (32 or 64 bits) located in folder *lib* is automatically loaded during import.

```
>>> from siliconsystems import *
```

### 2.2. Get Driver Information

The method *driver* can be used to obtain driver information. It does not expect any arguments and returns a named tuple with the following fields: *manufacturer* (manufacturing company), *version* (driver version) or *release* (driver release date, type *datetime.date*).
In the following example this information is requested and the output might be as follows.

```
>>> driver = Device.driver()
>>> driver
Driver(manufacturer='silicon systems', version='1.8.3', release=datetime.date(2016, 12, 21))
```

### 2.3. Configure Network Settings

Each device has the default IP address 192.168.1.1, the subnet mask 255.255.255.0, the gateway 0.0.0.0 and DHCP auto-configuration is enabled. If the name of the device or the network settings of a device shall be modified, e.g. DHCP auto-configuration needs to be disabled, method c*onfigure* may be used. Apart from the required serial number of the device, the device name (up to 64 characters), the DHCP status (either *Device.DHCP_ENABLED or Device.DHCP_DISABLED*), the host IP address, the subnet mask and the gateway IP address need to be specified. In order to restore the default configuration, method c*onfigure* can be called with the serial number only. The device needs to reside in the local area network (LAN) to be able to receive UDP broadcast datagrams.
In the following example for device with serial number 0x4704220B the device name is set to 'combustor monitor', DHCP auto-configuration is disabled, the host IP address is set to 192.168.100.20, the subnet mask is set to 255.255.255.0 and the gateway IP address is not utilized and set to 0.0.0.0. If the device is only accessed from within the local area network, the gateway is not required to be configured.

```
>>> Device.configure(0x4704220B, 'combustor monitor', Device.DHCP_DISABLED, '192.168.100.20',
'255.255.255.0', '0.0.0.0')
```

Below the configuration of the device with serial number 0x4704220B is reset.

```
>>> Device.configure(0x4704220B)
```

## 2.4. Finding Devices in the Local Area Network

If the IP address of the desired silicon systems devices are unknown, in particular, when the IP addresses are automatically assigned by a DHCP server, method *find* may be utilized. If the method is called without arguments, a vector containing all devices in the local area network is returned. The default time-out is 1000 ms and cannot be modified.

```
>>> device_list = Device.find()
```

The return value is a set of *Device* objects and, if e.g. two devices are found, might be as the follows.

```
>>> device_list
{<siliconsystems.Device.Device object at 0x01600CB0>, <siliconsystems.Device.Device object at 0x01600950>}
```

The number of devices can be determined with the *len* function.

```
>>> len(device_list)
2
```

If only the device with a certain serial number shall be looked up, method *find* can be called with one argument representing the serial number (unsigned integer). It returns the *Device* object, if successful.

```
>>> device = Device.find(0x4723BDF3)
>>> device
<siliconsystems.Device.Device object at 0x01EE7890>
```

If more devices shall be looked up at the same time, a set containing multiple serial numbers can be passed to the method. It returns a tuple containing serial numbers and device objects.

```
>>> device_list = Device.find({0x4723BDF3, 0x4723BDC2})
>>> device_list
{1193524674: <siliconsystems.Device.Device object at 0x01ECD770>, 1193524723: <siliconsystems.Device.Device
object at 0x01FBE0D0>}
```

## 2.5. Creating a Device Object

As described in the previous paragraph, method *find* returns a *Device* object or a list thereof, however a *Device* object can additionally be created manually if the IP address or the hostname is known. This avoids the invocation of method *find*.
In the following example the IP address is passed as a string and the return value might be as follows.

```
>>> device = Device('192.168.1.100')
>>> device
<siliconsystems.Device.Device object at 0x00148AB0>
```

If the DNS server in the LAN is able to resolve hostnames referring to local IP addresses, the hostname can be utilized instead of the IP address when creating a *Device* object.

```
>>> device = Device('VI01-472DA49F')
```

A *Device* object can also be created by passing an IP address of type *ipaddress.IPv4Address*. This requires the import of package *ipaddress*.

```
>>> import ipaddress
>>> address = ipaddress.IPv4Address('192.168.1.100')
>>> address
IPv4Address('192.168.1.100')
>>> device = Device(address)
```

In addition the hexadecimal representation of an IP address can be utilized to create a *Device* object.

```
>>> device = Device(0xC0A80164)
```

The copy constructor of class *Device* can be utilized to either duplicate an object or to create a derived object. This can particularly be useful when a derived object shall be created from a *Device* object returned by method *find*.
In the following example a *VI01* object is created from a *Device* object.

```
>>> device = Device.find(0x472DA49F)
>>> vi01 = VI01(device)
>>> vi01
<siliconsystems.VI01.VI01 object at 0x00F4D550>
```

## 2.6. Get Device IP Address

In order to get the address of a *Device* object method *address* is called without arguments. It returns a value of type *ipaddress.IPv4Address*.

```
>>> device = Device('192.168.1.100')
>>> device.address()
IPv4Address('192.168.1.100')
```

## 2.7. Set Device IP Address

The IP address of a *Device* object or any derived instance can also be modified by calling method *address* and passing the new IP address. Like for the creation of a *Device* object, several argument types may be utilized: the IP address as a string (e.g. '192.168.1.100'), the host name (e.g. 'VI01-472DA49F'), an *ipaddress.IPv4Address* type IP address or the hexadecimal representation of an IP address (e.g. 0x472DA49F).
In the following example only the first option is illustrated.

```
>>> device.address('192.168.1.100')
```

## 2.8. Get Device Time-out

When a command or request is sent to a device, a confirmation or response is expected within a certain time. If no response was received, a time-out error occurs. This maximum waiting time is determined by the time-out value which is 1000 ms by default and can be read out or set (in ms) with the aid of method *timeout* being called without arguments.
In the following example the current value is read out and the output might be as follows.

```
>>> device = Device('192.168.1.100')
>>> device.timeout()
1000
```

## 2.9. Set Device Time-out

In order to modify the time-out, the new value is passed to method *timeout*. A value of at least 1 ms is expected. In general, values lower than 100 ms are not recommended.
In the following example the time-out is set to 100 ms.

```
>>> device = Device('192.168.1.100')
>>> device.timeout(100)
```

## 2.10. Reset Device

Any device can be reset with method *reset*. The effect is the same as disconnecting and reconnecting the device from the Power over Ethernet (PoE) switch. This may be helpful when the network is being reconfigured (assignment of fixed IP addresses, etc.).
The method does not expect any arguments and is utilized as demonstrated below.

```
>>> device = Device('192.168.1.100')
>>> device.reset()
```

After the device is reset it takes several seconds until it reinitializes itself and gets a new IP address assigned by the DHCP server (if applicable). Until then the device cannot be accessed.

## 2.11. Acquisition of Device Information

With method *info* a variety of information can be requested from the device. The method does not expect any arguments and returns a named tuple *Device.Info* with the following fields: *model* (device model), *description* (functionality description), *manufacturer* (manufacturing company), *serial* (serial number), *revision* (hardware revision), *cpu* (CPU type), *frequency* (CPU frequency, in Hz), *memory* (random-access memory, in Bytes), *network* (network adapter), *version* (firmware version), *release* (firmware release date, *datetime.date*), *calibration* (calibration date, *datetime.date*), *expiration* (expiration of calibration date, *datetime.date*), *uptime reset* (time elapsed since reset or power on, in seconds), *uptime total* (total uptime, in seconds), *cycles* (number of power or reset cycles), *eth_host* (Ethernet address), *ip_host* (IP address, *ipaddress.IPv4Address*), *ip_subnet_mask* (subnet mask,

*ipaddress.IPv4Address*), *ip_gateway* (gateway IP address, *ipaddress.IPv4Address*), *dhcp* (DHCP status, either *Device.DHCP_ENABLED* or *Device.DHCP_DISABLED*), *hostname* (hostname for DNS look-up), *domain* (domain name), *name* (device name), *tag* (one-time programmable device tag).
In the following example this information is requested from the device and the output might be as follows.

```
>>> device = Device('192.168.1.100')
>>> info = device.info()
>>> info
Info(model='TMP02', description='Quad RTD Monitor', manufacturer='silicon systems', serial=1193524723,
revision='C', cpu='ATmega644A', frequency=20000000, memory=4096, network='CP2200', version='1.2.1',
release=datetime.date(2015, 10, 30), calibration=datetime.date(2016, 3, 8), expiration=datetime.date(2017,
3, 8), uptime_reset=16146, uptime_total=16146, cycles=45, eth_host='00:0B:3C:23:BD:F3',
ip_host=IPv4Address('192.168.1.100'), ip_subnet_mask=IPv4Address('255.255.255.0'),
ip_gateway=IPv4Address('192.168.1.1'), dhcp=0, hostname='TMP02-4723BDF3', domain='siliconsystems.at')
```

If the calibration or expiration of calibration dates is not applicable to the requested device, these fields are set to *None*. The fields of the named tuple *Device.Info* can be accessed using dot notation. In the following example the *manufacturer* field is retrieved.

```
>>> info.manufacturer
'silicon systems'
```

## 2.12. Acquisition of Device Status

Every device continuously monitors several parameters like supply voltages or the board temperature in order to ensure proper operation within specified conditions. These parameters can be retrieved with method *status* which does not expect any arguments and returns the named tuple *Device.Status* with two fields *property* and *ok*. Field *property* is a dictionary with pairs of symbol name and *Device.Property*. The named tuple *Device.Property* will be explained later on.
In the next example the status is requested from the device and the output might be as follows.

```
>>> device = Device('192.168.1.100')
>>> status = device.status()
>>> status.property
{'VREF': Property(symbol='VREF', description='Reference Voltage', unit='V', typical_value=5.0,
min_value=4.800000190734863, max_value=5.199999809265137, value=5.056250095367432), 'VCC':
Property(symbol='VCC', description='Positive Analog Supply Voltage', unit='V', typical_value=15.0,
min_value=14.0, max_value=16.0, value=14.490629196166992), 'VDD': Property(symbol='VDD',
description='Digital Supply Voltage', unit='V', typical_value=3.299999952316284,
min_value=3.0999999046325684, max_value=3.5, value=3.3423349857330322), 'VEE': Property(symbol='VEE',
description='Negative Analog Supply Voltage', unit='V', typical_value=-15.0, min_value=-16.0,
max_value=-14.0, value=-14.678053855895996), 'T': Property(symbol='T',
description='Board Temperature', unit='K', typical_value=nan, min_value=273.1499938964844,
max_value=343.1499938964844, value=310.8374938964844)}
```

The named tuple *Device.Property* has the following fields: *symbol* (symbol name), *description* (description of the parameter), *unit* (SI unit), *typical_value* (typical or nominal value, *math.nan* if not applicable), *min_value* (minimum value, *math.nan* if not applicable), *max_value* (maximum value, *math.nan* if not applicable), *value* (current value).
In the following example the current board temperature (symbol name 'T', in K) is retrieved. The return value might be as follows.

```
>>> status.property['T'].value
310.8374938964844
```

The *ok* flag indicates if all parameters are within their limits.

```
>>> status.ok
1
```

## 2.13. Burn Tag

Every device can store a non-volatile tag (up to 64 characters) which is one-time programmable. With method *burn* this tag can be programmed by the user and read out with method *info*.
In the following example the tag of the device at IP address 192.168.1.100 is set to 'dev_123'.

```
>>> device = Device('192.168.1.100')
>>> device.burn('dev_123')
```

# 3. CI01 Octal Current Monitor

The *CI01* device is a versatile and easy-to-use voltage monitor. With eight inputs, it can be used with any industrial transducer with the current output ranging from 4 to 20 mA. The ultra-low noise, the high resolution and the outstanding accuracy make it ideal for industrial applications as well as for scientific experiments. The channels are multiplexed, amplified, conditioned and sampled by the high-performance 24-Bit delta-sigma A/D converter.

## 3.1. Set Sampling Frequency

The sampling frequency of the A/D converter can be set with the aid of method *frequency*. All eight channels are sampled one after the other at the specified rate. The method expects one argument representing the sampling frequency. Valid values are *CI01.FREQUENCY_6*, *CI01.FREQUENCY_12*, *CI01.FREQUENCY_25*, *CI01.FREQUENCY_50*, *CI01.FREQUENCY_100*, *CI01.FREQUENCY_200*, *CI01.FREQUENCY_400*, *CI01.FREQUENCY_800*, *CI01.FREQUENCY_1500* and *CI01.FREQUENCY_3000* representing frequencies 6, 12, 25, 50, 100, 200, 400, 800, 1500 and 3000 Hz. By default the sampling frequency is 6 Hz to ensure lowest noise suitable for most applications.
This is illustrated in the following example where the frequency is set to 200 Hz.

```
>>> ci01 = CI01('192.168.1.103')
>>> ci01.frequency(CI01.FREQUENCY_200)
```

The sampling rate should not be set higher than necessary in order to keep the measurement noise as low as possible. Please refer to the data sheet for more details.

## 3.2. Measure Current

Method *measure* is utilized to acquire one or more samples from one or more channels (in A). Various call configurations do exist to suit the demands. If a single sample of all eight channels shall be acquired the method is called without arguments and returns a dictionary with pairs of channel number and current.
This is illustrated below and the return value might be as follows.

```
>>> ci01 = CI01('192.168.1.103')
>>> current = ci01.measure()
>>> current
{1: 0.014688492450667919, 2: 0.008467233772172334, 3: 0.01924146861722546, 4: 0.008741941040070682, 5:
0.01702027590326083, 6: 0.0073159732805764405, 7: 0.004703324210103526, 8: 0.007299539533897747}
```

If only one channel is to be sampled, its number (from 1 to 8) is passed to the method.
In the following example channel 2 is sampled and the return value might be as follows.

```
>>> ci01 = CI01('192.168.1.103')
>>> current = ci01.measure(2)
>>> current
0.011133613723794625
```

If more than one channel shall be sampled a set with the channel numbers must be passed to method *measure*. A dictionary with pairs of channel number and current is returned.
In the following example channels 1, 3 and 8 are sampled and the return value might be as follows.

```
>>> ci01 = CI01('192.168.1.103')
>>> current = ci01.measure({1, 3, 8})
>>> current
{1: 0.014688492450667919, 3: 0.01924146861722546, 8: 0.007299539533897747}
```

If more than one samples shall be acquired from one channel the method expects three arguments: the channel number, the number of samples (from 1 to $10^6$) and the sampling frequency (from 6 Hz to the sampling frequency configured with method *frequency*). A list of samples is returned.
In the next example at first the sampling frequency is set to 200 Hz and after that 10 samples from channel 4 are acquired at 180 Hz. The return value might be as follows.

```
>>> ci01 = CI01('192.168.1.103')
>>> ci01.frequency(CI01.FREQUENCY_200)
>>> current = ci01.measure(4, 10, 180)
>>> current
[0.014813688852808383, 0.014141591359651694, 0.011439864328871372, 0.007602226459165596,
0.005313371677419349, 0.017531524444149536, 0.005991667180959519, 0.012665059325667024, 0.01420250009021956]
```

Moreover the acquisition of several samples from several channels is possible. The expected arguments are similar to the previous call configuration: a set with the channel numbers, the number of samples (from 1 to $10^6$) and the sampling frequency (from 6 Hz to the sampling frequency configured with method *frequency*). A list of dictionaries with pairs of channel number and current is returned.
In the next example the sampling frequency is set to 200 Hz and 8 samples from channels 3 and 6 are acquired at 180 Hz. The return value might be as follows.

```
>>> ci01 = CI01('192.168.1.103')
>>> ci01.frequency(CI01.FREQUENCY_200)
>>> current = ci01.measure({3, 6}, 8, 180)
>>> current
[{3: 0.014724013081174945, 6: 0.01932148054104206}, {3: 0.009221456353032407, 6: 0.015095112248494398},
{3: 0.014060855333229866, 6: 0.016143454740056365}, {3: 0.01713786660365572, 6: 0.011774127549697115}, {3:
0.014489086496272728, 6: 0.015855569907573717}, {3: 0.013395149652840622, 6: 0.004774879591360671 5}, {3:
0.008214855323716396, 6: 0.01003909355816714}]
```

Indexing the list and the containing dictionaries is used to gather a specific sample. Note that indexing of lists is zero-based.
The example below shows how to retrieve sample 5 of channel 3 (compare to previous example).

```
>>> current[5 - 1][3]
0.014489086496272728
```

## 3.3. Stop Measurement

If method *measure* shall be terminated, method *stop* can be called, which is typically done from another thread or different computer.
The method does not expect any arguments and is utilized as demonstrated below.

```
>>> ci01 = CI01('192.168.1.103')
>>> ci01.stop()
```

# 4. CNT01 Quad Quadrature Encoder

The *CNT01* device is a versatile and easy-to-use quad absolute counter and quadrature encoder. Every counter supports the up/down counting mode to detect impulses of an arbitrary clock source as well as the quadrature encoder mode which is usually utilized to count the revolutions per minute of a spinning shaft or motor.

## 4.1. Set Counting Mode

In order to change the counting mode of one or more channels method *mode* is called. The following counting modes are supported: *CNT01.MODE_NORMAL* (normal, direction and clock signals), *CNT01.MODE_QUADRATURE_1X* (1x quadrature encoder), *CNT01.MODE_QUADRATURE_2X* (2x quadrature encoder) and *CNT01.MODE_QUADRATURE_4X* (4x quadrature encoder). Refer to the data sheet for more information. Various call configurations do exist to suit the demands. If the counting mode of all channels shall be updated only the counting mode is passed to the method.
In the following example all channels are configured for normal counting mode.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.mode(CNT01.MODE_NORMAL)
```

If the counting mode of only one channel is to be changed method *mode* is called with two arguments: the channel number (from 1 to 4) and the counting mode.
In the next example channel 2 is configured for 2x quadrature encoder counting.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.mode(2, CNT01.MODE_QUADRATURE_2X)
```

Additionally method *mode* can be utilized to configure several channels. In that case the method expects a set with the channel numbers as the first and the counting mode as the second argument.
Below channels 2 and 3 are configured for 4x quadrature encoder counting.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.mode({2, 3}, CNT01.MODE_QUADRATURE_4X)
```

Moreover method *mode* is able to configure several channels for different counting modes at the same time. A dictionary with pairs of channel number and mode is passed to the method.

In the following example channels 1, 2 and 4 are configured for normal, 1x quadrature encoder and 4x quadrature encoder counting.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.mode({1: CNT01.MODE_NORMAL, 2: CNT01.MODE_QUADRATURE_1X, 4: CNT01.MODE_QUADRATURE_4X})
```

## 4.2. Read from Channels

Method *read* is utilized to read the counter value of one or more channels. Various call configurations do exist to suit the demands. If the counter values of all four channels shall be acquired, method *read* is called without arguments and returns the named tuple *CNT01.Counter* with two fields *value* and *time*. The type of field *value* is a dictionary with pairs of channel number and counter value and field *time* is a relative timestamp (in ms) which can be utilized for precise frequency or rotational speed measurements.
This is demonstrated in the next example and the return value might be as follows.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> counter = cnt01.read()
>>> counter.value
{1: 1235, 2: 231, 3: 15689, 4: 0}
>>> counter.time
78018
```

If only the counter value of one channel is to be read out, the channel number (from 1 to 4) is passed to the method.
Channel 4 is read out as shown in the example below and the return value might be as follows.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> counter = cnt01.read(4)
>>> counter.value
18942
>>> counter.time
12097
```

If more than one channel shall be read out a set with the channel numbers must be passed to method *read*.
In the following example channels 1 and 3 are sampled and the return value might be as follows.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> counter = cnt01.read({1, 3})
>>> counter.value
{1: 42176, 3: 91574}
>>> counter.time
254894
```

## 4.3. Clear Channels

Method *clear* is utilized to reset the counter value of one or more channels to zero. If the counter values of all four channels shall be reset, method *clear* is called without arguments which is demonstrated in the example below.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.clear()
```

If the counter value of only one channel is to be reset, the channel number (from 1 to 4) is passed to method *clear*.
In the next example channel 1 is reset.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.clear(1)
```

Additionally, method *clear* can be utilized to reset the counter value of several channels to zero. In that case a set with the channel numbers is passed to the method.
In the example below channels 2 and 3 are reset.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.clear({2, 3})
```

## 4.4. Disable Channels

In order to disable one or more channels method *disable* is utilized. If all four channels shall be disabled, the method is called without arguments which is illustrated in the example below.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.disable()
```

If only one channel shall be disabled the channel number (from 1 to 4) is passed to method *disable*.
In the next example channel 2 is disabled.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.disable(2)
```

Additionally, method *disable* can be utilized to disable several channels. A set with the channel numbers is passed to the method.
In the example below channels 3 and 4 are disabled.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.disable({3, 4})
```

## 4.5. Enable Channels

One or more channels are enabled by the use of method *enable*. If all four channels shall be enabled, the method is called without arguments which is illustrated in the example below.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.enable()
```

If only one channel shall be enabled, the channel number (from 1 to 4) is passed to method *enable*.
In the next example channel 3 is enabled.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.enable(3)
```

Additionally, method *enable* can be utilized to enable several channels. In that case a set with the channel numbers is passed to the method.
In the example below channels 1 and 4 are enabled.

```
>>> cnt01 = CNT01('192.168.1.100')
>>> cnt01.enable({1, 4})
```

# 5. DIO01 Octal Bidirectional Digital I/O Device

The *DIO01* device is a versatile and easy-to-use octal, bidirectional digital I/O module. Every of the eight channels can individually be configured as input or output. Custom timers, counters, pulse generators, logic analyzers, functional tests or digital communication protocols like the widespread SPI bus system can easily be implemented. Digital control loops and custom serial or parallel protocols can be realized in software and modifications are done much more comfortable compared to equivalent hardware solutions.

## 5.1. Set Channel Direction

With the aid of method *direction* every channel can individually be configured. The following directions are supported: *DIO01.DIRECTION_INPUT* (input with pull-up resistors), *DIO01.DIRECTION_OUTPUT* (output). Various call configurations do exist to suit the demands. If the direction of all channels shall be changed the direction is passed to method *direction*. By default all eight channels are configured as input in order to protect any attached peripheral devices.
In the following example all channels are configured as output.

```
>>> dio01 = DIO01('192.168.1.101')
>>> dio01.direction(DIO01.DIRECTION_OUTPUT)
```

If the direction of only one channel is to be changed method *direction* is called with two arguments: the channel number (from 1 to 8) and the direction.
In the next example channel 2 is configured as input.

```
>>> dio01 = DIO01('192.168.1.101')
>>> dio01.direction(2, DIO01.DIRECTION_INPUT)
```

Additionally, method *direction* can be utilized to configure several channels. In that case the method expects a set with the channel numbers as the first and the direction as the second argument.
Below channels 2, 3 and 5 are configured as input.

```
>>> dio01 = DIO01('192.168.1.101')
>>> dio01.direction({2, 3, 5}, DIO01.DIRECTION_INPUT)
```

Moreover, method *direction* is able to configure several channels for different directions at the same time. It expects a dictionary with pairs of channel number and direction.
In the following example channels 1, 4 and 6 are set to input, input and output respectively.

```
>>> dio01 = DIO01('192.168.1.101')
>>> dio01.direction({1: DIO01.DIRECTION_INPUT, 4: DIO01.DIRECTION_INPUT, 6: DIO01.DIRECTION_OUTPUT})
```

If a channel is switched from input to output direction, its initial logic state is low.

## 5.2. Read from Channels

Method *read* is utilized to read the logic state of one or more channels which have been configured as input with method *direction*. The following logic states are possible: *DIO01.STATE_LOW* (logic low, *False*), *DIO01.STATE_HIGH* (logic high, *True*). Various call configurations do exist to suit the demands. If the logic states of all eight channels shall be acquired, method *read* is called without arguments and returns a dictionary with pairs of channel number and logic state.
This is demonstrated in the next example and the return value might be as follows.

```
>>> dio01 = DIO01('192.168.1.101')
>>> state = dio01.read()
>>> state
{1: 0, 2: 0, 3: 1, 4: 0, 5: 1, 6: 1, 7: 0, 8: 1}
```

If only the logic state of one channel is to be read out, the channel number (from 1 to 8) is passed to the method.
Channel 3 is read out as shown in the example below and the return value might be as follows.

```
>>> dio01 = DIO01('192.168.1.101')
>>> state = dio01.read(3)
>>> state
1
```

If more than one channel shall be read out, a set with the channel numbers must be passed to method *read*.
In the following example channels 2, 6 and 7 are sampled and the return value might be as follows.

```
>>> dio01 = DIO01('192.168.1.101')
>>> state = dio01.read({2, 6, 7})
>>> state
{2: 0, 6: 1, 7: 1}
```

## 5.3. Write to Channels

Method *write* allows to set one or more channels which have been configured as output with method *direction*. The following logic states are supported: *DIO01.STATE_LOW* (logic low, *False*), *DIO01.STATE_HIGH* (logic high, *True*). Various call configurations do exist to suit the demands. If the logic state of all channels shall be changed, the logic state is passed to method *write*.
In the following example all channels are set to logic high.

```
>>> dio01 = DIO01('192.168.1.101')
>>> dio01.write(DIO01.STATE_HIGH)
```

If only one channel is to be updated, method *write* is called with two arguments: the channel number (from 1 to 8) and the logic state.
In the next example channel 2 is set to logic low.

```
>>> dio01 = DIO01('192.168.1.101')
>>> dio01.write(2, DIO01.STATE_LOW)
```

Additionally, method *write* can be utilized to update several channels at the same time. In that case, the method expects a set with the channel numbers as the first and the logic state as the second argument.

Below channels 3, 4 and 5 are set to logic high.

```
>>> dio01 = DIO01('192.168.1.101')
>>> dio01.write({3, 4, 5}, DIO01.STATE_HIGH)
```

Moreover, method *write* is able to update several channels to different logic states at the same time. A dictionary with pairs of channel number and logic state is passed to the method.
In the following example channels 5, 7 and 8 are set to logic high, logic low and logic high.

```
>>> dio01 = DIO01('192.168.1.101')
>>> dio01.write({5: DIO01.STATE_HIGH, 7: DIO01.STATE_LOW, 8: DIO01.STATE_HIGH})
```

# 6. DIO02 Octal Bidirectional Digital I/O Device

The *DIO02* device is a versatile and easy-to-use octal, bidirectional digital I/O module. Every of the eight channels can individually be configured as input or output. Custom timers, counters, pulse generators, logic analyzers, functional tests or digital communication protocols like the widespread SPI bus system can easily be implemented. Digital control loops and custom serial or parallel protocols can be realized in software and modifications are done much more comfortable compared to equivalent hardware solutions.

## 6.1. Set Channel Direction

With the aid of method *direction* every channel can individually be configured. The following directions are supported: *DIO02.DIRECTION_INPUT* (input with pull-up resistors), *DIO02.DIRECTION_OUTPUT* (output). Various call configurations do exist to suit the demands. If the direction of all channels shall be changed the direction is passed to method *direction*. By default all eight channels are configured as input in order to protect any attached peripheral devices.
In the following example all channels are configured as output.

```
>>> dio02 = DIO02('192.168.1.109')
>>> dio02.direction(DIO02.DIRECTION_OUTPUT)
```

If the direction of only one channel is to be changed method *direction* is called with two arguments: the channel number (from 1 to 8) and the direction. In the next example channel 2 is configured as input.

```
>>> dio02 = DIO02('192.168.1.109')
>>> dio02.direction(2, DIO02.DIRECTION_INPUT)
```

Additionally, method *direction* can be utilized to configure several channels. In that case the method expects a set with the channel numbers as the first and the direction as the second argument.
Below channels 2, 3 and 5 are configured as input.

```
>>> dio02 = DIO02('192.168.1.109')
>>> dio02.direction({2, 3, 5}, DIO02.DIRECTION_INPUT)
```

Moreover, method *direction* is able to configure several channels for different directions at the same time. It expects a dictionary with pairs of channel number and direction.
In the following example channels 1, 4 and 6 are set to input, input and output respectively.

```
>>> dio02 = DIO02('192.168.1.109')
>>> dio02.direction({1: DIO02.DIRECTION_INPUT, 4: DIO02.DIRECTION_INPUT, 6: DIO02.DIRECTION_OUTPUT})
```

If a channel is switched from input to output direction, its initial logic state is low.

## 6.2. Read from Channels

Method *read* is utilized to read the logic state of one or more channels which have been configured as input with method *direction*. The following logic states are possible: *DIO02.STATE_LOW* (logic low, *False*), *DIO02.STATE_HIGH* (logic high, *True*). Various call configurations do exist to suit the demands. If the logic states of all eight channels shall be acquired, method *read* is called without arguments and returns a dictionary with pairs of channel number and logic state.
This is demonstrated in the next example and the return value might be as follows.

```
>>> dio02 = DIO02('192.168.1.109')
>>> state = dio02.read()
>>> state
{1: 0, 2: 0, 3: 1, 4: 0, 5: 1, 6: 1, 7: 0, 8: 1}
```

If only the logic state of one channel is to be read out, the channel number (from 1 to 8) is passed to the method.
Channel 3 is read out as shown in the example below and the return value might be as follows.

```
>>> dio02 = DIO02('192.168.1.109')
>>> state = dio02.read(3)
>>> state
1
```

If more than one channel shall be read out, a set with the channel numbers must be passed to method *read*.
In the following example channels 2, 6 and 7 are sampled and the return value might be as follows.

```
>>> dio02 = DIO02('192.168.1.109')
>>> state = dio02.read({2, 6, 7})
>>> state
{2: 0, 6: 1, 7: 1}
```

## 6.3. Write to Channels

Method *write* allows to set one or more channels which have been configured as output with method *direction*. The following logic states are supported: *DIO02.STATE_LOW* (logic low, *False*), *DIO02.STATE_HIGH* (logic high, *True*). Various call configurations do exist to suit the demands. If the logic state of all channels shall be changed, the logic state is passed to method *write*.
In the following example all channels are set to logic high.

```
>>> dio02 = DIO02('192.168.1.109')
>>> dio02.write(DIO02.STATE_HIGH)
```

If only one channel is to be updated, method *write* is called with two arguments: the channel number (from 1 to 8) and the logic state.
In the next example channel 2 is set to logic low.

```
>>> dio02 = DIO02('192.168.1.109')
>>> dio02.write(2, DIO02.STATE_LOW)
```

Additionally, method *write* can be utilized to update several channels at the same time. In that case, the method expects a set with the channel numbers as the first and the logic state as the second argument.
Below channels 3, 4 and 5 are set to logic high.

```
>>> dio02 = DIO02('192.168.1.109')
>>> dio02.write({3, 4, 5}, DIO02.STATE_HIGH)
```

Moreover, method *write* is able to update several channels to different logic states at the same time. A dictionary with pairs of channel number and logic state is passed to the method.
In the following example channels 5, 7 and 8 are set to logic high, logic low and logic high.

```
>>> dio02 = DIO02('192.168.1.109')
>>> dio02.write({5: DIO02.STATE_HIGH, 7: DIO02.STATE_LOW, 8: DIO02.STATE_HIGH})
```

# 7. DO01 Octal Digital Output Buffer

The *DO01* device is a versatile and easy-to-use octal high-current digital output module. Every of the eight channels can individually be set to logic low or high. Depending on the voltage rating of the connected transducers, an external power supply is needed to supply them. Electro-mechanical relays, electrical or electro-pneumatic valves, long transmission lines, digital transducers or even DC motors can directly be interfaced with the *DO01* device.

## 7.1. Write to Channels

Method *write* allows to set one or more channels. The following logic states are supported: *DO01.STATE_LOW* (logic low, *False*, no voltage), *DO01.STATE_HIGH* (logic high, *True*, supply voltage). By default the state of all eight channels is logic low in order to keep the outputs unpowered. Various call configurations do exist to suit the demands. If the logic state of all channels shall be changed, the logic state is passed to method *write*.
In the following example all channels are set to logic high.

```
>>> do01 = DO01('192.168.1.102')
>>> do01.write(DO01.STATE_HIGH)
```

If only one channel is to be updated, method *write* is called with two arguments: the channel number (from 1 to 8) and the logic state.
In the next example channel 3 is set to logic low.

```
>>> do01 = DO01('192.168.1.102')
>>> do01.write(3, DO01.STATE_LOW)
```

Additionally, method *write* can be utilized to update several channels. In that case, the method expects a set with the channel numbers as the first and the logic state as the second argument.
Below channels 1, 2 and 7 are set to logic high.

```
>>> do01 = DO01('192.168.1.102')
>>> do01.write({1, 2, 7}, DO01.STATE_HIGH)
```

Method *write* is able to update several channels to different logic states at the same time. A dictionary with pairs of channel number and logic state is passed to the method.
In the following example channels 2, 4 and 8 are set to logic low, logic low and logic high.

```
>>> do01 = DO01('192.168.1.102')
>>> do01.write({2: DO01.STATE_LOW, 4: DO01.STATE_LOW, 8: DO01.STATE_HIGH})
```

## 7.2. Measure Current

The *DO01* device is able to measure the current (in A) through all eight channels to provide useful feedback information. With the aid of method *measure* the current of one or more channels can be read out. If the currents of all eight channels shall be acquired, the method must be called without arguments and it returns a dictionary with pairs of channel number and current.
This is demonstrated in the next example and the return value might be as follows.

```
>>> do01 = DO01('192.168.1.102')
>>> current = do01.measure()
>>> current
{1: 0.11986352797975819, 2: 1.4488937333706404, 3: 1.7505335022611566, 4: 1.5798000941072539, 5:
0.4301272272638017, 6: 0.2643992204559915, 7: 1.7594437810109043, 8: 1.5939713473845725}
```

If only the current of one channel is to be acquired, the channel number (from 1 to 8) is passed to the method.
Channel 4 is read out as shown in the example below and the return value might be as follows.

```
>>> do01 = DO01('192.168.1.102')
>>> current = do01.measure(4)
>>> current
1.0099062420986085
```

If more than one channel shall be read out, a set with the channel numbers must be passed to method *measure*.
In the following example channels 1, 4 and 6 are sampled and the return value might be as follows.

```
>>> do01 = DO01('192.168.1.102')
>>> current = do01.measure({1, 4, 6})
>>> current
{1: 1.2689130450877042, 4: 0.9713729719732345, 6: 0.6823609868143321}
```

# 8. DO02 Octal SPST Relay Module

The *DO02* device is a octal easy-to-use high-voltage high-current relay module. Each channel can be enabled or disabled individually. Electrical or electro-pneumatic valves, heaters or AC / DC motors can be controlled with the *DO02* device.

## 8.1. Write to Channels

Method *write* allows to set one or more relays. The following states are supported: *DO02.STATE_OPEN* (open contact, *False*), *DO02.STATE_CLOSED* (closed contact, *True*). By default the state of all relays is open contact in order to keep the attached components unpowered, if applicable. Various call configurations do exist to suit the demands. If the state of all relays shall be changed, it is passed to method *write*.
In the following example all relays are closed.

```
>>> do02 = DO02('192.168.1.108')
>>> do02.write(DO02.STATE_CLOSED)
```

If only one relay is to be updated, method *write* is called with two arguments: the channel number (from 1 to 8) and the state.
In the next example relay 2 is opened.

```
>>> do02 = DO02('192.168.1.108')
>>> do02.write(2, DO02.STATE_OPEN)
```

Additionally, method *write* can be utilized to update several relays. In that case, the method expects a set with the channel numbers as the first and the state as the second argument.
Below relays 2, 3 and 4 are closed.

```
>>> do02 = DO02('192.168.1.108')
>>> do02.write({2, 3, 4}, DO02.STATE_CLOSED)
```

Method *write* is able to update several relays to different states at the same time. A dictionary with pairs of channel number and state is passed to the method.
In the following example relays 1, 2 and 7 are closed, closed and opened respectively.

```
>>> do02 = DO02('192.168.1.108')
>>> do02.write({1: DO02.STATE_CLOSED, 2: DO02.STATE_CLOSED, 7: DO02.STATE_OPEN})
```

# 9. TMP01 Octal Thermocouple Monitor

The *TMP01* device is a versatile and easy-to-use temperature monitor. With eight inputs, it can be used with nearly any thermocouple type. The device was designed to meet the demands of scientific or industrial applications where the high temperature range, ultra-low noise and high resolution are important concerns.

## 9.1. Temperature Sensors

The class *TMP01.Sensor* represents a temperature sensor. When an object is created, the sensor type can be passed to the constructor. The following sensor types are supported: *TMP01.Sensor.TYPE_B*, *TMP01.Sensor.TYPE_C*, *TMP01.Sensor.TYPE_E*, *TMP01.Sensor.TYPE_J*, *TMP01.Sensor.TYPE_K*, *TMP01.Sensor.TYPE_M*, *TMP01.Sensor.TYPE_N*, *TMP01.Sensor.TYPE_P*, *TMP01.Sensor.TYPE_R*, *TMP01.Sensor.TYPE_S*, *TMP01.Sensor.TYPE_T* representing thermocouple types B, C, E, J, K, M, N, P, R, S, T. In addition, special sensor types *TMP01.Sensor.TYPE_CUSTOM* (custom temperature sensor, method *measure* returns the sampled voltage) and *TMP01.Sensor.TYPE_NONE* (no sensor) are defined.
The following example shows the creation of a thermocouple type J.

```
>>> sensor = TMP01.Sensor(TMP01.Sensor.TYPE_J)
```

If no sensor type is passed to the constructor, the sensor type is set to *TMP01.Sensor.TYPE_NONE* as shown in the example below.

```
>>> sensor = TMP01.Sensor()
```

The type of sensor can be read out or modified by the use of method *type*. If no arguments are passed to the method, it returns the sensor type.
Below the sensor type is retrieved and the return value might be as follows.

```
>>> sensor = TMP01.Sensor(TMP01.Sensor.TYPE_K)
>>> sensor.type()
6
```

If the sensor type is to be modified, the sensor type is passed to method *type*.
In the following example an object with thermocouple type K is created and then the sensor type is modified to a custom sensor.

```
>>> sensor = TMP01.Sensor(TMP01.Sensor.TYPE_K)
>>> sensor.type(TMP01.Sensor.TYPE_CUSTOM)
>>> sensor.type()
1
```

## 9.2. Set Sampling Frequency

The sampling frequency of the A/D converter can be set with the aid of method *frequency*. All eight channels are sampled one after the other at the specified rate. The method expects one argument representing the sampling frequency. The following frequencies are supported: *TMP01.FREQUENCY_6*, *TMP01.FREQUENCY_12*, *TMP01.FREQUENCY_25*, *TMP01.FREQUENCY_50*, *TMP01.FREQUENCY_100*, *TMP01.FREQUENCY_200*, *TMP01.FREQUENCY_400*, *TMP01.FREQUENCY_800*, *TMP01.FREQUENCY_1500* and *TMP01.FREQUENCY_3000* representing frequencies 6, 12, 25, 50, 100, 200, 400, 800, 1500 and 3000 Hz. By default the sampling frequency is 6 Hz to ensure lowest noise suitable for most applications.
This is illustrated in the following example where the frequency is set to 100 Hz.

```
>>> tmp01 = TMP01('192.168.1.105')
>>> tmp01.frequency(TMP01.FREQUENCY_100)
```

The sampling rate should not be set higher than necessary in order to keep the measurement noise as low as possible. Please refer to the data sheet for more details.

## 9.3. Measure Temperature

Method *measure* is utilized to read the temperature from one or more channels (in K). Various call configurations do exist to suit the demands. If all eight channels shall be sampled, the method is be called without arguments and it returns a dictionary with pairs of channel number and temperature. The return value of the next example might be as follows.

```
>>> tmp01 = TMP01('192.168.1.105')
>>> temperature = tmp01.measure()
>>> temperature
{1: 368.67038562427996, 2: 311.8024112091409, 3: 344.6008825272939, 4: 331.10072741663697, 5:
368.2362869166435, 6: 303.6834403374732, 7: 331.67740711569263, 8: 311.81128610668736}
```

If only one channel is to be sampled, its number (from 1 to 8) is passed to the method.
In the example below channel 3 is read out and the return value might be as follows.

```
>>> tmp01 = TMP01('192.168.1.105')
>>> temperature = tmp01.measure(3)
>>> temperature
311.8024112091409
```

If more than one channel shall be sampled, a set with the channel numbers must be passed to method *measure*.
In the following example channels 2, 5 and 7 are sampled and the return value might be as follows.

```
>>> tmp01 = TMP01('192.168.1.105')
>>> temperature = tmp01.measure({2, 5, 7})
>>> temperature
{2: 311.8024112091409, 5: 368.2362869166435, 7: 331.67740711569263}
```

## 9.4. Set Temperature Sensor

With the use of method *sensor* every channel can individually be configured to be used with a different type of temperature sensor. If all channels of a device shall be utilized with the same type of sensor the sensor object (type *TMP01.Sensor*) is passed to method *sensor*. By default all channels are disabled in order to avoid unintentional misconfiguration.
In the following example all channels are set to thermocouple type K.

```
>>> tmp01 = TMP01('192.168.1.105')
>>> sensor = TMP01.Sensor(TMP01.Sensor.TYPE_K)
>>> tmp01.sensor(sensor)
```

If one or more channels are not utilized, these channels should be configured as sensor type *TMP01.Sensor.TYPE_NONE* which increases the effective sampling rate of the remaining channels. If only one channel is to be configured, method *sensor* is called with two arguments: the channel number (from 1 to 8) and the sensor object.
In the next example channel 3 is configured for custom temperature sensors.

```
>>> tmp01 = TMP01('192.168.1.105')
>>> sensor = TMP01.Sensor(TMP01.Sensor.TYPE_CUSTOM)
>>> tmp01.sensor(3, sensor)
```

Additionally, method *sensor* can be utilized to configure several channels for the same sensor type. A set with the channel numbers as the first and the sensor object as the second argument are passed to the method.
Below channels 1, 2 and 7 are disabled.

```
>>> tmp01 = TMP01('192.168.1.105')
>>> sensor = TMP01.Sensor(TMP01.Sensor.TYPE_NONE)
>>> tmp01.sensor({1, 2, 7}, sensor)
```

Moreover, method *sensor* is able to configure several channels to be utilized with different temperature sensors at the same time. In that case, the method expects a dictionary with pairs of channel number and sensor object.
In the next example channels 2 and 8 are set to thermocouple type J and T respectively.

```
>>> tmp01 = TMP01('192.168.1.105')
>>> sensor1 = TMP01.Sensor(TMP01.Sensor.TYPE_J)
>>> sensor2 = TMP01.Sensor(TMP01.Sensor.TYPE_T)
>>> tmp01.sensor({2: sensor1, 8: sensor2})
```

## 10. TMP02 Quad RTD Monitor

The *TMP02* device is a versatile and easy-to-use temperature monitor. With four inputs, it can be used with Platinum resistors, temperature diodes or NTC thermistors. The device was designed to meet the demands of scientific or industrial applications where the high temperature range, ultra-low noise and high resolution are important concerns.

### 10.1. Temperature Sensors

The class *TMP02.Sensor* represents a temperature sensor. When an object is created, the sensor type can be passed to the constructor. The following sensor types are supported: *TMP02.Sensor.TYPE_PT*, *TMP02.Sensor.TYPE_DT470*, *TMP02.Sensor.TYPE_DT670*, *TMP02.Sensor.TYPE_KTY81_1*, *TMP02.Sensor.TYPE_KTY81_2*, *TMP02.Sensor.TYPE_KTY82_1*, *TMP02.Sensor.TYPE_KTY82_2*, *TMP02.Sensor.TYPE_KTY83_1*, *TMP02.Sensor.TYPE_KTY84_1*. Refer to the datasheet for additional information on the sensor types. In addition, special sensor types *TMP02.Sensor.TYPE_CUSTOM* (custom temperature sensor, method *measure* returns the sampled voltage) and *TMP02.Sensor.TYPE_NONE* (no sensor) are defined.
The following example shows the creation of a KTY1-1 type (normalized resistance of 1 kΩ at 25 °C) NTC temperature sensor.

```
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_KTY81_1)
```

For platinum type PTC temperature sensors, the normalized resistance (defined at 0 °C) can be passed to the constructor as the second argument. If it is omitted, the default value of 100 Ω is used. Normalized resistances between 1 Ω and 1 kΩ are supported.
Below the creation of a PT-500 temperature sensor is illustrated.

```
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_PT, 500)
```

Though for all temperature sensors, the voltage range and the excitation current can be specified, this feature is particularly useful for custom temperature sensors. The following voltage ranges are supported: *TMP02.Sensor.VOLTAGE_0V5* (0.5 V), *TMP02.Sensor.VOLTAGE_1V* (1 V), *TMP02.Sensor.VOLTAGE_2V5* (2.5 V), *TMP02.Sensor.VOLTAGE_5V* (5 V), *TMP02.Sensor.VOLTAGE_DEFAULT* (default voltage range, sensor type dependant). These excitation currents can be selected: *TMP02.Sensor.CURRENT_10UA* (10 µA), *TMP02.Sensor.CURRENT_1MA* (1 mA), *TMP02.Sensor.CURRENT_DEFAULT* (default excitation current, sensor type dependant). In addition to the sensor type, the voltage range and the excitation current can be passed to the constructor.
In the following example the low temperature diode DT-670 with the voltage range of 2.5 V and the excitation current of 10 µA is created.

```
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_DT670, TMP02.Sensor.VOLTAGE_2V5, TMP02.Sensor.CURRENT_10UA)
```

The type of sensor can be read out or modified by the use of method *type*. If no arguments are passed to the method, it returns the sensor type.
Below the sensor type is retrieved and the return value might be as follows.

```
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_KTY81_1)
>>> sensor.type()
5
```

If the sensor type is to be modified, the sensor type is passed to method *type*.
In the following example a PT-500 temperature sensor is created and then the sensor type is modified to a custom sensor. The return value might be as follows.

```
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_PT)
>>> sensor.type(TMP02.Sensor.TYPE_CUSTOM)
>>> sensor.type()
1
```

If the normalized resistance of the sensor shall be read out or modified, method *resistance* is used. If no arguments are passed to the method, it returns the normalized resistance.
Below the normalized resistance of a PT temperature sensor is retrieved.

```
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_PT)
>>> sensor.resistance()
100.0
```

If the normalized resistance is to be modified, the normalized resistance is passed to method *resistance*.
In the following example a PT-100 temperature sensor is created and then the normalized resistance is changed to 500 Ω.

```
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_PT)
>>> sensor.resistance(500)
```

With method *voltage*, the voltage range can be retrieved or modified. If no arguments are passed to the method, the voltage range is returned.
The following example shows how to retrieve the voltage range.

```
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_DT670, TMP02.Sensor.VOLTAGE_2V5, TMP02.Sensor.CURRENT_10UA)
>>> sensor.voltage()
2.5
```

In order to modify the voltage range, it is passed to method voltage.
In the following example the 1.0 V voltage range is selected.

```
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_PT)
>>> sensor.voltage(TMP02.Sensor.VOLTAGE_1V)
```

It is also possible to read out or to modify the excitation current. This can be accomplished with method *current*. If the method is called without arguments, it returns the excitation current.
The example below shows how to read out the excitation current.

```
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_DT470, TMP02.Sensor.VOLTAGE_5V, TMP02.Sensor.CURRENT_10UA)
>>> sensor.current()
1e-05
```

In the following example the excitation current is reset to default.

```
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_DT470, TMP02.Sensor.VOLTAGE_5V, TMP02.Sensor.CURRENT_10UA)
>>> sensor.current(TMP02.Sensor.CURRENT_DEFAULT)
```

## 10.2. Set Sampling Frequency

The sampling frequency of the A/D converter can be set with the aid of method *frequency*. All eight channels are sampled one after the other at the specified rate. The method expects one argument representing the sampling frequency. The following frequencies are supported: *TMP02.FREQUENCY_6*, *TMP02.FREQUENCY_12*, *TMP02.FREQUENCY_25*, *TMP02.FREQUENCY_50*, *TMP02.FREQUENCY_100*, *TMP02.FREQUENCY_200*, *TMP02.FREQUENCY_400*, *TMP02.FREQUENCY_800*, *TMP02.FREQUENCY_1500* and *TMP02.FREQUENCY_3000* representing frequencies 6, 12, 25, 50, 100, 200, 400, 800, 1500 and 3000 Hz. By default the sampling frequency is 6 Hz to ensure lowest noise suitable for most applications.
This is illustrated in the following example where the frequency is set to 50 Hz.

```
>>> tmp02 = TMP02('192.168.1.106')
>>> tmp02.frequency(TMP02.FREQUENCY_50)
```

The sampling rate should not be set higher than necessary in order to keep the measurement noise as low as possible. Please refer to the data sheet for more details.

## 10.3. Measure Temperature

Method *measure* is utilized to read the temperature from one or more channels (in K). Various call configurations do exist to suit the demands. If all eight channels shall be sampled, the method is be called without arguments and it returns a dictionary with pairs of channel number and temperature. The return value of the next example might be as follows.

```
>>> tmp02 = TMP02('192.168.1.106')
>>> temperature = tmp02.measure()
>>> temperature
{1: 368.67038562427996, 2: 311.8024112091409, 3: 344.6008825272939, 4: 331.10072741663697}
```

If only one channel is to be sampled, its number (from 1 to 4) is passed to the method.
In the example below channel 3 is read out and the return value might be as follows.

```
>>> tmp02 = TMP02('192.168.1.106')
>>> temperature = tmp02.measure(3)
>>> temperature
311.8024112091409
```

If more than one channel shall be sampled, a set with the channel numbers must be passed to method *measure*.
In the following example channels 1, 3 and 4 are sampled and the return value might be as follows.

```
>>> tmp02 = TMP02('192.168.1.105')
>>> temperature = tmp02.measure({1, 3, 4})
>>> temperature
{1: 311.8024112091409, 3: 368.2362869166435, 4: 331.67740711569263}
```

## 10.4. Set Temperature Sensor

With the use of method *sensor* every channel can individually be configured to be used with a different type of temperature sensor. If all channels of a device shall be utilized with the same type of sensor the sensor object (type *TMP02.Sensor*) is passed to method *sensor*. By default all channels are disabled in order to avoid unintentional misconfiguration.
In the following example all channels are set to PT-100 temperature sensor.

```
>>> tmp02 = TMP02('192.168.1.106')
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_PT)
>>> tmp02.sensor(sensor)
```

If one or more channels are not utilized, these channels should be configured as sensor type *TMP02.Sensor.TYPE_NONE* which increases the effective sampling rate of the remaining channels. If only one channel is to be configured, method *sensor* is called with two arguments: the channel number (from 1 to 4) and the sensor object.
In the next example channel 3 is configured for custom temperature sensors.

```
>>> tmp02 = TMP02('192.168.1.106')
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_CUSTOM)
>>> tmp02.sensor(3, sensor)
```

Additionally, method *sensor* can be utilized to configure several channels for the same sensor type. A set with the channel numbers as the first and the sensor object as the second argument are passed to the method.
Below channels 1, 2 and 4 are disabled.

```
>>> tmp02 = TMP02('192.168.1.106')
>>> sensor = TMP02.Sensor(TMP02.Sensor.TYPE_NONE)
>>> tmp02.sensor({1, 2, 4}, sensor)
```

Moreover, method *sensor* is able to configure several channels to be utilized with different temperature sensors at the same time. In that case, the method expects a dictionary with pairs of channel number and sensor object.
In the next example channels 2 and 3 are set to low temperature diode DT-470 and PT-100 temperature sensor respectively.

```
>>> tmp02 = TMP02('192.168.1.106')
>>> sensor1 = TMP02.Sensor(TMP02.Sensor.TYPE_DT470)
>>> sensor2 = TMP02.Sensor(TMP02.Sensor.TYPE_PT)
>>> tmp02.sensor({2: sensor1, 3: sensor2})
```

# 11. VI01 Octal Voltage Monitor

The *VI01* device is a versatile and easy-to-use voltage monitor. With eight inputs, it can be used with any industrial transducer with the voltage output ranging from -10 V to +10 V. The ultra-low noise, the high resolution and the outstanding accuracy make it ideal for industrial applications as well as for scientific experiments. The channels are multiplexed, amplified, conditioned and sampled by the high-performance 24-Bit delta-sigma A/D converter.

## 11.1. Set Sampling Frequency

The sampling frequency of the A/D converter can be set with the aid of method *frequency*. All eight channels are sampled one after the other at the specified rate. The method expects one argument representing the sampling frequency. Valid values are *VI01.FREQUENCY_6*, *VI01.FREQUENCY_12*, *VI01.FREQUENCY_25*, *VI01.FREQUENCY_50*, *VI01.FREQUENCY_100*, *VI01.FREQUENCY_200*, *VI01.FREQUENCY_400*, *VI01.FREQUENCY_800*, *VI01.FREQUENCY_1500* and *VI01.FREQUENCY_3000* representing frequencies 6, 12, 25, 50, 100, 200, 400, 800, 1500 and 3000 Hz. By default the sampling frequency is 6 Hz to ensure lowest noise suitable for most applications.
This is illustrated in the following example where the frequency is set to 400 Hz.

```
>>> vi01 = VI01('192.168.1.107')
>>> vi01.frequency(VI01.FREQUENCY_400)
```

The sampling rate should not be set higher than necessary in order to keep the measurement noise as low as possible. Please refer to the data sheet for more details.

## 11.2. Measure Voltage

Method *measure* is utilized to acquire one or more samples from one or more channels (in V). Various call configurations do exist to suit the demands. If a single sample of all eight channels shall be acquired the method is called without arguments and returns a dictionary with pairs of channel number and voltage.
This is illustrated below and the return value might be as follows.

```
>>> vi01 = VI01('192.168.1.107')
>>> voltage = vi01.measure()
>>> voltage
{1: -0.5972805455760266, 2: 8.774007583078888, 3: 8.218679151958344, 4: 2.57064558783547, 5:
9.866371435779875, 6: 2.343150916541159, 7: 0.4396105026580983, 8: -1.5683471211771138}
```

If only one channel is to be sampled, its number (from 1 to 8) is passed to the method.
In the following example channel 6 is sampled and the return value might be as follows.

```
>>> vi01 = VI01('192.168.1.107')
>>> voltage = vi01.measure(6)
>>> voltage
5.047992680958544
```

If more than one channel shall be sampled a set with the channel numbers must be passed to method *measure*. A dictionary with pairs of channel number and voltage is returned.
In the following example channels 2, 4 and 7 are sampled and the return value might be as follows.

```
>>> vi01 = VI01('192.168.1.107')
>>> voltage = vi01.measure({2, 4, 7})
>>> voltage
{2: -5.907751008404361, 4: 3.4847245555623534, 7: -4.683837645655773}
```

If more than one samples shall be acquired from one channel the method expects three arguments: the channel number, the number of samples (from 1 to $10^6$) and the sampling frequency (from 6 Hz to the sampling frequency configured with method *frequency*). A list of samples is returned.
In the next example at first the sampling frequency is set to 800 Hz and after that 10 samples from channel 3 are acquired at 800 Hz. The return value might be as follows.

```
>>> vi01 = VI01('192.168.1.107')
>>> vi01.frequency(VI01.FREQUENCY_800)
>>> voltage = vi01.measure(3, 10, 800)
>>> voltage
[5.0035584878000074, 5.000416991002845, 5.0015453843932365, 5.0098226782852855, 5.003863160195338,
5.001879859132699, 5.002806026165695, 5.006832483742139, 5.008882836260877, 5.008555696721807]
```

Moreover the acquisition of several samples from several channels is possible. The expected arguments are similar to the previous call configuration: a set with the channel numbers, the number of samples (from 1 to $10^6$) and the sampling frequency (from 6 Hz to the sampling frequency configured with method *frequency*). A list of dictionaries with pairs of channel number and voltage is returned.
In the next example the sampling frequency is set to 200 Hz and 7 samples from channels 2 and 3 are acquired at 190 Hz. The return value might be as follows.

```
>>> vi01 = VI01('192.168.1.107')
>>> vi01.frequency(VI01.FREQUENCY_200)
>>> voltage = vi01.measure({2, 3}, 7, 190)
>>> voltage
[{2: 5.001426978746452, 3: 3.0095688390916457}, {2: 5.001228846480992, 3: 3.0032225176017415},
{2: 5.00257503349055, 3: 3.005348418826106}, {2: 5.0022511677552925, 3: 3.0036813740799135}, {2:
5.002188428721803, 3: 3.0073717744516446}, {2: 5.007661141962965, 3: 3.007279341761345}, {2:
5.00026614454084, 3: 3.0062490408303058}]
```

Indexing the list and the containing dictionaries is used to gather a specific sample. Note that indexing of lists is zero-based.
The example below shows how to retrieve sample 4 of channel 2 (compare to previous example).

```
>>> voltage[4 - 1][3]
5.0022511677552925
```

### 11.3. Stop Measurement

If method *measure* shall be terminated, method *stop* can be called, which is typically done from another thread or different computer.
The method does not expect any arguments and is utilized as demonstrated below.

```
>>> vi01 = VI01('192.168.1.107')
>>> vi01.stop()
```

## 12. VO01 Octal Voltage Output Device

The *VO01* device is a versatile and easy-to-use voltage output device. With eight outputs, it can be used with any analog industrial interface with voltage input ranging from -10 V to +10 V. The ultra-low noise, the high resolution and the outstanding accuracy make it ideal for industrial applications as well as for scientific experiments.

### 12.1. Control the Output Voltage

In order to control the output voltage of one or more channels the method *control* is utilized. If all eight channels of a device shall be modified the new voltage (from -10 V to +10 V) is passed to the method *control*. By default all channels are set to 0 V to protect any attached peripheral devices.
In the following example all channels are set to +3.2 V.

```
>>> vo01 = VO01('192.168.1.104')
>>> vo01.control(3.2)
```

If only one channel is to be modified method *control* is called with two arguments: the channel number (from 1 to 8) and the voltage (from -10 V to +10 V).
In the next example channel 3 is set to -2.4 V.

```
>>> vo01 = VO01('192.168.1.104')
>>> vo01.control(3, -2.4)
```

Additionally, method *control* can be utilized to set several channels to a common voltage. In that case, the method expects a set with the channel numbers as the first and the voltage as the second argument.
Below channels 1, 2 and 7 are set to +6.8 V.

```
>>> vo01 = VO01('192.168.1.104')
>>> vo01.control({1, 2, 7}, 6.8)
```

Moreover, method *control* is able to change the voltages of several channels to different values at the same time. In this case, the method expects a dictionary with pais of channel number and voltage.
In the following example channels 2, 4 and 8 are set to +2.1 V, -5.3 V and +8.1 V respectively.

```
>>> vo01 = VO01('192.168.1.104')
>>> vo01.control({2: 2.1, 4: -5.3, 8: 8.1})
```

## 13. Exception Handling

If one of methods described in that document could not be executed properly an exception is thrown which shall be caught by the caller of the method. The reason of the exception can be deduced from the caught object. Please refer to the Python programming reference for more information on exception handling techniques.

The following exceptions are derived from class *Error*: *Error.Aborted* (raised when e.g. a measurement requests are interrupted), *Error.Configuration* (raised when the device is not properly configured, e.g. when an unconfigured channel shall be read out), *Error.Failed* (raised when a request failed due to an unknown reason), *Error.Internal* (raised when an internal unhandled exception occurs), *Error.Network* (raised when a communication problem occurred), *Error.NotFound* (raised when method *Device.find* could not find one or more devices), *Error.Parameter* (raised when a passed parameter is invalid or out of range), *Error.Reserved* (raised when e.g. a measurement request is pending), *Error.Timeout* (raised when no response has been received from a device).

In the following example the voltage of channel 9 of device *VI01* shall be measured, but only eight channels are available. This causes an *Error.Parameter* exception. If no try/except blocks are utilized, the output might be as follows.

```
>>> vi01 = VI01('192.168.0.1')
>>> voltage = vi01.measure(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "d:\siliconsystems\VI01.py", line 93, in measure
    self._error(error)
  File "d:\siliconsystems\Device.py", line 425, in _error
    raise Device._error_[error]
siliconsystems.Error.Parameter: One or more parameters are out of range or invalid. Check the validity of
all parameters.
```

In the following example the operation is attempted within a try/except block. The error message is printed and might be as follows.

```
>>> try:
...     vi01 = VI01('192.168.0.1')
...     vi01.measure(9)
... except Error.Error as error:
...     print(error)
...
One or more parameters are out of range or invalid. Check the validity of all parameters.
```

Moreover, it is possible to handle specific exceptions.

The following example shows how to specifically handle any network related errors. Other errors are commonly caught.

```
>>> try:
...     vi01 = VI01('192.168.0.1')
...     vi01.measure(1)
... except Error.Network as error:
...     print(error)
... except Error.Error as error:
...     print(error)
```

In order to investigate a raised exception, library *traceback* can be utilized. Below the call stack is printed when an *Error.Timeout* exception occurs.

Status: April 2022