



## PYTHON API Reference



# Table of contents

<b>1. Introduction .....</b>	<b>1</b>
<b>2. Using the Yocto-Demo with Python .....</b>	<b>3</b>
2.1. Source files .....	3
2.2. Dynamic library .....	3
2.3. Control of the Led function .....	3
2.4. Control of the module part .....	5
2.5. Error handling .....	7
Blueprint .....	10
<b>3. Reference .....</b>	<b>10</b>
3.1. General functions .....	11
3.2. Accelerometer function interface .....	32
3.3. AnButton function interface .....	74
3.4. CarbonDioxide function interface .....	112
3.5. ColorLed function interface .....	151
3.6. Compass function interface .....	180
3.7. Current function interface .....	220
3.8. DataLogger function interface .....	259
3.9. Formatted data sequence .....	290
3.10. Recorded data sequence .....	300
3.11. Unformatted data sequence .....	312
3.12. Digital IO function interface .....	327
3.13. Display function interface .....	371
3.14. DisplayLayer object interface .....	418
3.15. External power supply control interface .....	450
3.16. Files function interface .....	475
3.17. GenericSensor function interface .....	503
3.18. Gyroscope function interface .....	549
3.19. Yocto-hub port interface .....	600
3.20. Humidity function interface .....	625
3.21. Led function interface .....	664
3.22. LightSensor function interface .....	691
3.23. Magnetometer function interface .....	731
3.24. Measured value .....	773

3.25. Module control interface .....	779
3.26. Network function interface .....	822
3.27. OS control .....	879
3.28. Power function interface .....	902
3.29. Pressure function interface .....	945
3.30. Pwm function interface .....	984
3.31. PwmPowerSource function interface .....	1022
3.32. Quaternion interface .....	1045
3.33. Real Time Clock function interface .....	1084
3.34. Reference frame configuration .....	1111
3.35. Relay function interface .....	1147
3.36. Sensor function interface .....	1183
3.37. Servo function interface .....	1222
3.38. Temperature function interface .....	1257
3.39. Tilt function interface .....	1298
3.40. Voc function interface .....	1337
3.41. Voltage function interface .....	1376
3.42. Voltage source function interface .....	1415
3.43. WakeUpMonitor function interface .....	1443
3.44. WakeUpSchedule function interface .....	1478
3.45. Watchdog function interface .....	1515
3.46. Wireless function interface .....	1560
<b>Index .....</b>	<b>1589</b>

# 1. Introduction

This manual is intended to be used as a reference for Yoctopuce Python library, in order to interface your code with USB sensors and controllers.

The next chapter is taken from the free USB device Yocto-Demo, in order to provide a concrete examples of how the library is used within a program.

The remaining part of the manual is a function-by-function, class-by-class documentation of the API. The first section describes all general-purpose global function, while the forthcoming sections describe the various classes that you may have to use depending on the Yoctopuce device being used. For more informations regarding the purpose and the usage of a given device attribute, please refer to the extended discussion provided in the device-specific user manual.



## 2. Using the Yocto-Demo with Python

Python is an interpreted object oriented language developed by Guido van Rossum. Among its advantages is the fact that it is free, and the fact that it is available for most platforms, Windows as well as UNIX. It is an ideal language to write small scripts on a napkin. The Yoctopuce library is compatible with Python 2.6+ and 3+. It works under Windows, Mac OS X, and Linux, Intel as well as ARM. The library was tested with Python 2.6 and Python 3.2. Python interpreters are available on the Python web site<sup>1</sup>.

### 2.1. Source files

The Yoctopuce library classes<sup>2</sup> for Python that you will use are provided as source files. Copy all the content of the *Sources* directory in the directory of your choice and add this directory to the *PYTHONPATH* environment variable. If you use an IDE to program in Python, refer to its documentation to configure it so that it automatically finds the API source files.

### 2.2. Dynamic library

A section of the low-level library is written in C, but you should not need to interact directly with it: it is provided as a DLL under Windows, as a .so files under UNIX, and as a .dylib file under Mac OS X. Everything was done to ensure the simplest possible interaction from Python: the distinct versions of the dynamic library corresponding to the distinct operating systems and architectures are stored in the *cdll* directory. The API automatically loads the correct file during its initialization. You should not have to worry about it.

If you ever need to recompile the dynamic library, its complete source code is located in the Yoctopuce C++ library.

In order to keep them simple, all the examples provided in this documentation are console applications. Naturally, the libraries function in a strictly identical manner if you integrate them in an application with a graphical interface.

### 2.3. Control of the Led function

A few lines of code are enough to use a Yocto-Demo. Here is the skeleton of a Python code snippet to use the Led function.

---

<sup>1</sup> <http://www.python.org/download/>

<sup>2</sup> [www.yoctopuce.com/EN/libraries.php](http://www.yoctopuce.com/EN/libraries.php)

```
[...]
errmsg=YRefParam()
#Get access to your device, connected locally on USB for instance
YAPI.RegisterHub("usb",errmsg)
led = YLed.FindLed("YCTOPOC1-123456.led")

# Hot-plug is easy: just check that the device is online
if led.isOnline():
    #Use led.set_power()
    ...
[...]
```

Let's look at these lines in more details.

## YAPI.RegisterHub

The `yAPI.RegisterHub` function initializes the Yoctopuce API and indicates where the modules should be looked for. When used with the parameter "`usb`", it will use the modules locally connected to the computer running the library. If the initialization does not succeed, this function returns a value different from `YAPI.SUCCESS` and `errmsg` contains the error message.

## YLed.FindLed

The `YLed.FindLed` function allows you to find a led from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-Demo module with serial number `YCTOPOC1-123456` which you have named "`MyModule`", and for which you have given the `led` function the name "`MyFunction`". The following five calls are strictly equivalent, as long as "`MyFunction`" is defined only once.

```
led = YLed.FindLed("YCTOPOC1-123456.led")
led = YLed.FindLed("YCTOPOC1-123456.MyFunction")
led = YLed.FindLed("MyModule.led")
led = YLed.FindLed("MyModule.MyFunction")
led = YLed.FindLed("MyFunction")
```

`YLed.FindLed` returns an object which you can then use at will to control the led.

## isOnline

The `isOnline()` method of the object returned by `YLed.FindLed` allows you to know if the corresponding module is present and in working order.

## set\_power

The `set_power()` function of the objet returned by `YLed.FindLed` allows you to turn on and off the led. The argument is `YLed.POWER_ON` or `YLed.POWER_OFF`. In the reference on the programming interface, you will find more methods to precisely control the luminosity and make the led blink automatically.

## A real example

Launch Python and open the corresponding sample script provided in the directory **Examples/Doc-GettingStarted-Yocto-Demo** of the Yoctopuce library.

In this example, you will recognize the functions explained above, but this time used with all side materials needed to make it work nicely as a small demo.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import os,sys
from yocto_api import *
from yocto_led import *

def usage():
    scriptname = os.path.basename(sys.argv[0])
    print("Usage:")
```

```

print(scriptname+' <serial_number>')
print(scriptname+' <logical_name>')
print(scriptname+' any ')
sys.exit()

def die(msg):
    sys.exit(msg+' (check USB cable)')

def setLedState(led,state):
    if led.isOnline():
        if state :
            led.set_power(YLed.POWER_ON)
        else:
            led.set_power(YLed.POWER_OFF)
    else:
        print('Module not connected (check identification and USB cable)')

errmsg=YRefParam()

if len(sys.argv)<2 : usage()

target=sys.argv[1]

# Setup the API to use local USB devices
if YAPI.RegisterHub("usb", errmsg)!= YAPI.SUCCESS:
    sys.exit("init error"+errmsg.value)

if target=='any':
    # retreive any RGB led
    led = YLed.FirstLed()
    if led is None :
        die('No module connected')
else:
    led= YLed.FindLed(target + '.led')

if not(led.isOnline()):die('device not connected')

print('0: turn test led OFF')
print('1: turn test led ON')
print('x: exit')

try: input = raw_input # python 2.x fix
except: pass

c= input("command:")

while c!='x':
    if c=='0' : setLedState(led,False);
    elif c=='1' :setLedState(led,True);
    c= input("command:")

```

## 2.4. Control of the module part

Each module can be controlled in a similar manner, you can find below a simple sample program displaying the main parameters of the module and enabling you to activate the localization beacon.

```

#!/usr/bin/python
# -*- coding: utf-8 -*-
import os,sys
from yocto_api import *

def usage():
    sys.exit("usage: demo <serial or logical name> [ON/OFF]")

errmsg =YRefParam()
if YAPI.RegisterHub("usb", errmsg) != YAPI.SUCCESS:
    sys.exit("RegisterHub error: " + str(errmsg))

if len(sys.argv)<2 : usage()

m = YModule.FindModule(sys.argv[1]) ## use serial or logical name

```

```

if m.isOnline():
    if len(sys.argv) > 2:
        if sys.argv[2].upper() == "ON" : m.set_beacon(YModule.BEACON_ON)
        if sys.argv[2].upper() == "OFF" : m.set_beacon(YModule.BEACON_OFF)

    print("serial:      " + m.get_serialNumber())
    print("logical name: " + m.get_logicalName())
    print("luminosity:   " + str(m.get_luminosity()))
    if m.get_beacon() == YModule.BEACON_ON:
        print("beacon:      ON")
    else:
        print("beacon:      OFF")
    print("upTime:       " + str(m.get_upTime()/1000)+" sec")
    print("USB current:  " + str(m.get_usbCurrent())+" mA")
    print("logs:\n" + m.get_lastLogs())
else:
    print(sys.argv[1] + " not connected (check identification and USB cable)")

```

Each property `xxx` of the module can be read thanks to a method of type `YModule.get_xxxx()`, and properties which are not read-only can be modified with the help of the `YModule.set_xxx()` method. For more details regarding the used functions, refer to the API chapters.

## Changing the module settings

When you want to modify the settings of a module, you only need to call the corresponding `YModule.set_xxx()` function. However, this modification is performed only in the random access memory (RAM) of the module: if the module is restarted, the modifications are lost. To memorize them persistently, it is necessary to ask the module to save its current configuration in its permanent memory. To do so, use the `YModule.saveToFlash()` method. Inversely, it is possible to force the module to forget its current settings by using the `YModule.revertFromFlash()` method. The short example below allows you to modify the logical name of a module.

```

#!/usr/bin/python
# -*- coding: utf-8 -*-
import os,sys
from yocto_api import *

def usage():
    sys.exit("usage: demo <serial or logical name> <new logical name>")

if len(sys.argv) != 3 : usage()

errormsg =YRefParam()
if YAPI.RegisterHub("usb", errormsg) != YAPI.SUCCESS:
    sys.exit("RegisterHub error: " + str(errormsg))

m = YModule.FindModule(sys.argv[1]) # use serial or logical name

if m.isOnline():
    newname = sys.argv[2]
    if not YAPI.CheckLogicalName(newname):
        sys.exit("Invalid name (" + newname + ")")
    m.set_logicalName(newname)
    m.saveToFlash() # do not forget this
    print ("Module: serial= " + m.get_serialNumber() + " / name= " + m.get_logicalName())
else:
    sys.exit("not connected (check identification and USB cable)")

```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guarantees that the saving process is performed correctly. This limit, linked to the technology employed by the module micro-processor, is located at about 100000 cycles. In short, you can use the `YModule.saveToFlash()` function only 100000 times in the life of the module. Make sure you do not call this function within a loop.

## List the modules

Obtaining the list of the connected modules is performed with the `YModule.yFirstModule()` function which returns the first module found. Then, you only need to call the `nextModule()` function of this object to find the following modules, and this as long as the returned value is not null. Below a short example listing the connected modules.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import os,sys

from yocto_api import *

errmsg=YRefParam()

# Setup the API to use local USB devices
if YAPI.RegisterHub("usb", errmsg)!= YAPI.SUCCESS:
    sys.exit("init error"+str(errmsg))

print('Device list')

module = YModule.FirstModule()
while module is not None:
    print(module.get_serialNumber()+' ('+module.get_productName()+')')
    module = module.nextModule()
```

## 2.5. Error handling

When you implement a program which must interact with USB modules, you cannot disregard error handling. Inevitably, there will be a time when a user will have unplugged the device, either before running the software, or even while the software is running. The Yoctopuce library is designed to help you support this kind of behavior, but your code must nevertheless be conceived to interpret in the best possible way the errors indicated by the library.

The simplest way to work around the problem is the one used in the short examples provided in this chapter: before accessing a module, check that it is online with the `isOnline` function, and then hope that it will stay so during the fraction of a second necessary for the following code lines to run. This method is not perfect, but it can be sufficient in some cases. You must however be aware that you cannot completely exclude an error which would occur after the call to `isOnline` and which could crash the software. The only way to prevent this is to implement one of the two error handling techniques described below.

The method recommended by most programming languages for unpredictable error handling is the use of exceptions. By default, it is the behavior of the Yoctopuce library. If an error happens while you try to access a module, the library throws an exception. In this case, there are three possibilities:

- If your code catches the exception and handles it, everything goes well.
- If your program is running in debug mode, you can relatively easily determine where the problem happened and view the explanatory message linked to the exception.
- Otherwise... the exception makes your program crash, bang!

As this latest situation is not the most desirable, the Yoctopuce library offers another possibility for error handling, allowing you to create a robust program without needing to catch exceptions at every line of code. You simply need to call the `yDisableExceptions()` function to commute the library to a mode where exceptions for all the functions are systematically replaced by specific return values, which can be tested by the caller when necessary. For each function, the name of each return value in case of error is systematically documented in the library reference. The name always follows the same logic: a `get_state()` method returns a `Y_STATE_INVALID` value, a `get_currentValue` method returns a `Y_CURRENTVALUE_INVALID` value, and so on. In any case, the returned value is of the expected type and is not a null pointer which would risk crashing your program. At worst, if you display the value without testing it, it will be outside the expected bounds for the returned value. In the case of functions which do not normally return information, the return value is `YAPI_SUCCESS` if everything went well, and a different error code in case of failure.

When you work without exceptions, you can obtain an error code and an error message explaining the source of the error. You can request them from the object which returned the error, calling the `errType()` and `errMessage()` methods. Their returned values contain the same information as in the exceptions when they are active.



### **3. Reference**

## 3.1. General functions

These general functions should be used to initialize and configure the Yoctopuce library. In most cases, a simple call to function `yRegisterHub()` should be enough. The module-specific functions `yFind...()` or `yFirst...()` should then be used to retrieve an object that provides interaction with the module.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_api.js'></script>
node.js var yoctolib = require('yoctolib');
var YAPI = yoctolib.YAPI;
var YModule = yoctolib.YModule;
php require_once('yocto_api.php');
cpp #include "yocto_api.h"
m #import "yocto_api.h"
pas uses yocto_api;
vb yocto_api.vb
cs yocto_api.cs
java import com.yoctopuce.YoctoAPI.YModule;
py from yocto_api import *

```

### Global functions

#### `yCheckLogicalName(name)`

Checks if a given string is valid as logical name for a module or a function.

#### `yDisableExceptions()`

Disables the use of exceptions to report runtime errors.

#### `yEnableExceptions()`

Re-enables the use of exceptions for runtime error handling.

#### `yEnableUSBHost(osContext)`

This function is used only on Android.

#### `yFreeAPI()`

Frees dynamically allocated memory blocks used by the Yoctopuce library.

#### `yGetAPIVersion()`

Returns the version identifier for the Yoctopuce library in use.

#### `yGetTickCount()`

Returns the current value of a monotone millisecond-based time counter.

#### `yHandleEvents(errmsg)`

Maintains the device-to-library communication channel.

#### `yInitAPI(mode, errmsg)`

Initializes the Yoctopuce programming library explicitly.

#### `yPreregisterHub(url, errmsg)`

Fault-tolerant alternative to RegisterHub().

#### `yRegisterDeviceArrivalCallback(arrivalCallback)`

Register a callback function, to be called each time a device is plugged.

#### `yRegisterDeviceRemovalCallback(removalCallback)`

Register a callback function, to be called each time a device is unplugged.

#### `yRegisterHub(url, errmsg)`

Setup the Yoctopuce library to use modules connected on a given machine.

#### `yRegisterHubDiscoveryCallback(hubDiscoveryCallback)`

### 3. Reference

Register a callback function, to be called each time an Network Hub send an SSDP message.

#### **yRegisterLogFunction(logfun)**

Registers a log callback function.

#### **ySelectArchitecture(arch)**

Select the architecture or the library to be loaded to access to USB.

#### **ySetDelegate(object)**

(Objective-C only) Register an object that must follow the protocol YDeviceHotPlug.

#### **ySetTimeout(callback, ms\_timeout, arguments)**

Invoke the specified callback function after a given timeout.

#### **ySleep(ms\_duration, errmsg)**

Pauses the execution flow for a specified duration.

#### **yTriggerHubDiscovery(errmsg)**

Force a hub discovery, if a callback as been registered with yRegisterDeviceRemovalCallback it will be called for each net work hub that will respond to the discovery.

#### **yUnregisterHub(url)**

Setup the Yoctopuce library to no more use modules connected on a previously registered machine with RegisterHub.

#### **yUpdateDeviceList(errmsg)**

Triggers a (re)detection of connected Yoctopuce modules.

#### **yUpdateDeviceList\_async(callback, context)**

Triggers a (re)detection of connected Yoctopuce modules.

**YAPI.CheckLogicalName()****YAPI****yCheckLogicalName()YAPI.CheckLogicalName()**

Checks if a given string is valid as logical name for a module or a function.

```
def CheckLogicalName( name)
```

A valid logical name has a maximum of 19 characters, all among A..Z, a..z, 0..9, \_, and -. If you try to configure a logical name with an incorrect string, the invalid characters are ignored.

**Parameters :**

**name** a string containing the name to check.

**Returns :**

`true` if the name is valid, `false` otherwise.

## **YAPI.DisableExceptions()**

**YAPI**

### **yDisableExceptions()YAPI.DisableExceptions()**

---

Disables the use of exceptions to report runtime errors.

```
def DisableExceptions( )
```

When exceptions are disabled, every function returns a specific error value which depends on its type and which is documented in this reference manual.

**YAPI.EnableExceptions()****YAPI****yEnableExceptions()YAPI.EnableExceptions()**

Re-enables the use of exceptions for runtime error handling.

```
def EnableExceptions( )
```

Be aware than when exceptions are enabled, every function that fails triggers an exception. If the exception is not caught by the user code, it either fires the debugger or aborts (i.e. crash) the program. On failure, throws an exception or returns a negative error code.

## **YAPI.FreeAPI() yFreeAPI()YAPI.FreeAPI()**

---

**YAPI**

Frees dynamically allocated memory blocks used by the Yoctopuce library.

```
def FreeAPI( )
```

It is generally not required to call this function, unless you want to free all dynamically allocated memory blocks in order to track a memory leak for instance. You should not call any other library function after calling `yFreeAPI( )`, or your program will crash.

## YAPI.GetAPIVersion() yGetAPIVersion()YAPI.GetAPIVersion()

YAPI

Returns the version identifier for the Yoctopuce library in use.

```
def GetAPIVersion( )
```

The version is a string in the form "Major.Minor.Build", for instance "1.01.5535". For languages using an external DLL (for instance C#, VisualBasic or Delphi), the character string includes as well the DLL version, for instance "1.01.5535 (1.01.5439)".

If you want to verify in your code that the library version is compatible with the version that you have used during development, verify that the major number is strictly equal and that the minor number is greater or equal. The build number is not relevant with respect to the library compatibility.

**Returns :**

a character string describing the library version.

## YAPI.GetTickCount() yGetTickCount()YAPI.GetTickCount()

YAPI

Returns the current value of a monotone millisecond-based time counter.

```
def GetTickCount( )
```

This counter can be used to compute delays in relation with Yoctopuce devices, which also uses the millisecond as timebase.

**Returns :**

a long integer corresponding to the millisecond counter.

**YAPI.HandleEvents()****YAPI****yHandleEvents()YAPI.HandleEvents()**

Maintains the device-to-library communication channel.

```
def HandleEvents( errmsg=None)
```

If your program includes significant loops, you may want to include a call to this function to make sure that the library takes care of the information pushed by the modules on the communication channels. This is not strictly necessary, but it may improve the reactivity of the library for the following commands.

This function may signal an error in case there is a communication problem while contacting a module.

**Parameters :**

**errmsg** a string passed by reference to receive any error message.

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## YAPI.InitAPI() yInitAPI()YAPI.InitAPI()

YAPI

Initializes the Yoctopuce programming library explicitly.

```
def InitAPI( mode, errmsg=None)
```

It is not strictly needed to call `yInitAPI()`, as the library is automatically initialized when calling `yRegisterHub()` for the first time.

When `Y_DETECT_NONE` is used as detection mode, you must explicitly use `yRegisterHub()` to point the API to the VirtualHub on which your devices are connected before trying to access them.

### Parameters :

`mode` an integer corresponding to the type of automatic device detection to use. Possible values are `Y_DETECT_NONE`, `Y_DETECT_USB`, `Y_DETECT_NET`, and `Y_DETECT_ALL`.

`errmsg` a string passed by reference to receive any error message.

### Returns :

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## YAPI.PreregisterHub() yPreregisterHub()YAPI.PreregisterHub()

YAPI

Fault-tolerant alternative to RegisterHub().

```
def PreregisterHub( url, errmsg=None)
```

This function has the same purpose and same arguments as RegisterHub( ), but does not trigger an error when the selected hub is not available at the time of the function call. This makes it possible to register a network hub independently of the current connectivity, and to try to contact it only when a device is actively needed.

### Parameters :

**url** a string containing either "usb", "callback" or the root URL of the hub to monitor  
**errmsg** a string passed by reference to receive any error message.

### Returns :

YAPI\_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

**YAPI.RegisterDeviceArrivalCallback()**  
**yRegisterDeviceArrivalCallback()**  
**YAPI.RegisterDeviceArrivalCallback()****YAPI**

Register a callback function, to be called each time a device is plugged.

```
def RegisterDeviceArrivalCallback( arrivalCallback )
```

This callback will be invoked while `yUpdateDeviceList` is running. You will have to call this function on a regular basis.

**Parameters :**

**arrivalCallback** a procedure taking a `YModule` parameter, or null

**YAPI.RegisterDeviceRemovalCallback()**  
**yRegisterDeviceRemovalCallback()**  
**YAPI.RegisterDeviceRemovalCallback()****YAPI**

Register a callback function, to be called each time a device is unplugged.

```
def RegisterDeviceRemovalCallback( removalCallback )
```

This callback will be invoked while `yUpdateDeviceList` is running. You will have to call this function on a regular basis.

**Parameters :**

`removalCallback` a procedure taking a `YModule` parameter, or null

## YAPI.RegisterHub()

YAPI

### yRegisterHub()YAPI.RegisterHub()

Setup the Yoctopuce library to use modules connected on a given machine.

```
def RegisterHub( url, errmsg=None)
```

The parameter will determine how the API will work. Use the following values:

**usb**: When the **usb** keyword is used, the API will work with devices connected directly to the USB bus. Some programming languages such as Javascript, PHP, and Java don't provide direct access to USB hardware, so **usb** will not work with these. In this case, use a VirtualHub or a networked YoctoHub (see below).

**x.x.x.x** or **hostname**: The API will use the devices connected to the host with the given IP address or hostname. That host can be a regular computer running a VirtualHub, or a networked YoctoHub such as YoctoHub-Ethernet or YoctoHub-Wireless. If you want to use the VirtualHub running on your local computer, use the IP address 127.0.0.1.

**callback**: This keyword makes the API run in "*HTTP Callback*" mode. This is a special mode allowing to take control of Yoctopuce devices through a NAT filter when using a VirtualHub or a networked YoctoHub. You only need to configure your hub to call your server script on a regular basis. This mode is currently available for PHP and Node.js only.

Be aware that only one application can use direct USB access at a given time on a machine. Multiple access would cause conflicts while trying to access the USB modules. In particular, this means that you must stop the VirtualHub software before starting an application that uses direct USB access. The workaround for this limitation is to setup the library to use the VirtualHub rather than direct USB access.

If access control has been activated on the hub, virtual or not, you want to reach, the URL parameter should look like:

```
http://username:password@adresse:port
```

You can call *RegisterHub* several times to connect to several machines.

#### Parameters :

- url** a string containing either "usb", "callback" or the root URL of the hub to monitor
- errmsg** a string passed by reference to receive any error message.

#### Returns :

YAPI\_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

## YAPI.RegisterHubDiscoveryCallback() yRegisterHubDiscoveryCallback() YAPI.RegisterHubDiscoveryCallback()

YAPI

Register a callback function, to be called each time an Network Hub send an SSDP message.

```
def RegisterHubDiscoveryCallback( hubDiscoveryCallback)
```

The callback has two string parameter, the first one contain the serial number of the hub and the second contain the URL of the network hub (this URL can be passed to RegisterHub). This callback will be invoked while yUpdateDeviceList is running. You will have to call this function on a regular basis.

### Parameters :

**hubDiscoveryCallback** a procedure taking two string parameter, or null

**YAPI.RegisterLogFunction()****YAPI****yRegisterLogFunction()YAPI.RegisterLogFunction()**

Registers a log callback function.

```
def RegisterLogFunction( logfun )
```

This callback will be called each time the API have something to say. Quite useful to debug the API.

**Parameters :**

**logfun** a procedure taking a string parameter, or null

**YAPI.SelectArchitecture()****YAPI****ySelectArchitecture()YAPI.SelectArchitecture()**

Select the architecture or the library to be loaded to access to USB.

```
def SelectArchitecture( arch)
```

By default, the Python library automatically detects the appropriate library to use. However, for Linux ARM, it is not possible to reliably distinguish between a Hard Float (armhf) and a Soft Float (armel) install. For this case, it is therefore recommended to manually select the proper architecture by calling `SelectArchitecture()` before any other call to the library.

**Parameters :**

**arch** A string containing the architecture to use. Possible values are: "armhf", "armel", "i386", "x86\_64", "32bit", "64bit"

**Returns :**

nothing.

On failure, throws an exception.

## YAPI.Sleep() ySleep()YAPI.Sleep()

YAPI

Pauses the execution flow for a specified duration.

```
def Sleep( ms_duration, errmsg=None)
```

This function implements a passive waiting loop, meaning that it does not consume CPU cycles significantly. The processor is left available for other threads and processes. During the pause, the library nevertheless reads from time to time information from the Yoctopuce modules by calling `yHandleEvents()`, in order to stay up-to-date.

This function may signal an error in case there is a communication problem while contacting a module.

### Parameters :

`ms_duration` an integer corresponding to the duration of the pause, in milliseconds.  
`errmsg` a string passed by reference to receive any error message.

### Returns :

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

**YAPI.TriggerHubDiscovery()****YAPI****yTriggerHubDiscovery()YAPI.TriggerHubDiscovery()**

Force a hub discovery, if a callback has been registered with yRegisterDeviceRemovalCallback it will be called for each network hub that will respond to the discovery.

```
def TriggerHubDiscovery( errmsg=None)
```

**Parameters :**

**errmsg** a string passed by reference to receive any error message.

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## YAPI.UnregisterHub() yUnregisterHub()YAPI.UnregisterHub()

YAPI

Setup the Yoctopuce library to no more use modules connected on a previously registered machine with RegisterHub.

```
def UnregisterHub( url)
```

### Parameters :

**url** a string containing either "usb" or the

**YAPI.UpdateDeviceList()****YAPI****yUpdateDeviceList()YAPI.UpdateDeviceList()**

Triggers a (re)detection of connected Yoctopuce modules.

```
def UpdateDeviceList( errmsg=None)
```

The library searches the machines or USB ports previously registered using `yRegisterHub()`, and invokes any user-defined callback function in case a change in the list of connected devices is detected.

This function can be called as frequently as desired to refresh the device list and to make the application aware of hot-plug events.

**Parameters :**

`errmsg` a string passed by reference to receive any error message.

**Returns :**

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

## 3.2. Accelerometer function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_accelerometer.js'></script>
nodejs var yoctolib = require('yoctolib');
var YAccelerometer = yoctolib.YAccelerometer;
php require_once('yocto_accelerometer.php');
cpp #include "yocto_accelerometer.h"
m #import "yocto_accelerometer.h"
pas uses yocto_accelerometer;
vb yocto_accelerometer.vb
cs yocto_accelerometer.cs
java import com.yoctopuce.YoctoAPI.YAccelerometer;
py from yocto_accelerometer import *

```

### Global functions

#### **yFindAccelerometer(func)**

Retrieves an accelerometer for a given identifier.

#### **yFirstAccelerometer()**

Starts the enumeration of accelerometers currently accessible.

### YAccelerometer methods

#### **accelerometer→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### **accelerometer→describe()**

Returns a short text that describes unambiguously the instance of the accelerometer in the form TYPE (NAME) = SERIAL . FUNCTIONID.

#### **accelerometer→get\_advertisedValue()**

Returns the current value of the accelerometer (no more than 6 characters).

#### **accelerometer→get\_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### **accelerometer→get\_currentValue()**

Returns the current value of the acceleration.

#### **accelerometer→get\_errorMessage()**

Returns the error message of the latest error with the accelerometer.

#### **accelerometer→get\_errorType()**

Returns the numerical error code of the latest error with the accelerometer.

#### **accelerometer→get\_friendlyName()**

Returns a global identifier of the accelerometer in the format MODULE\_NAME . FUNCTION\_NAME.

#### **accelerometer→get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### **accelerometer→get\_functionId()**

Returns the hardware identifier of the accelerometer, without reference to the module.

#### **accelerometer→get\_hardwareId()**

Returns the unique hardware identifier of the accelerometer in the form SERIAL . FUNCTIONID.

<b>accelerometer→get_highestValue()</b>	Returns the maximal value observed for the acceleration since the device was started.
<b>accelerometer→get_logFrequency()</b>	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>accelerometer→get_logicalName()</b>	Returns the logical name of the accelerometer.
<b>accelerometer→get_lowestValue()</b>	Returns the minimal value observed for the acceleration since the device was started.
<b>accelerometer→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>accelerometer→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>accelerometer→get_recordedData(startTime, endTime)</b>	Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>accelerometer→get_reportFrequency()</b>	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>accelerometer→get_resolution()</b>	Returns the resolution of the measured values.
<b>accelerometer→get_unit()</b>	Returns the measuring unit for the acceleration.
<b>accelerometer→get(userData)</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>accelerometer→get_xValue()</b>	Returns the X component of the acceleration, as a floating point number.
<b>accelerometer→get_yValue()</b>	Returns the Y component of the acceleration, as a floating point number.
<b>accelerometer→get_zValue()</b>	Returns the Z component of the acceleration, as a floating point number.
<b>accelerometer→isOnline()</b>	Checks if the accelerometer is currently reachable, without raising any error.
<b>accelerometer→isOnline_async(callback, context)</b>	Checks if the accelerometer is currently reachable, without raising any error (asynchronous version).
<b>accelerometer→load(msValidity)</b>	Preloads the accelerometer cache with a specified validity duration.
<b>accelerometer→loadCalibrationPoints(rawValues, refValues)</b>	Retrieves error correction data points previously entered using the method calibrateFromPoints.
<b>accelerometer→load_async(msValidity, callback, context)</b>	Preloads the accelerometer cache with a specified validity duration (asynchronous version).
<b>accelerometer→nextAccelerometer()</b>	Continues the enumeration of accelerometers started using yFirstAccelerometer( ).
<b>accelerometer→registerTimedReportCallback(callback)</b>	Registers the callback function that is invoked on every periodic timed notification.
<b>accelerometer→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.

### 3. Reference

---

**accelerometer→set\_highestValue(newval)**

Changes the recorded maximal value observed.

**accelerometer→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**accelerometer→set\_logicalName(newval)**

Changes the logical name of the accelerometer.

**accelerometer→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**accelerometer→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**accelerometer→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**accelerometer→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**accelerometer→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YAccelerometer.FindAccelerometer() yFindAccelerometer() YAccelerometer.FindAccelerometer()

## YAccelerometer

Retrieves an accelerometer for a given identifier.

```
def FindAccelerometer( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the accelerometer is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAccelerometer.isOnline()` to test if the accelerometer is indeed online at a given time. In case of ambiguity when looking for an accelerometer by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the accelerometer

### Returns :

a `YAccelerometer` object allowing you to drive the accelerometer.

## **YAccelerometer.FirstAccelerometer()**

## **YAccelerometer**

### **yFirstAccelerometer()**

### **YAccelerometer.FirstAccelerometer()**

---

Starts the enumeration of accelerometers currently accessible.

```
def FirstAccelerometer( )
```

Use the method `YAccelerometer.nextAccelerometer()` to iterate on next accelerometers.

#### **Returns :**

a pointer to a `YAccelerometer` object, corresponding to the first accelerometer currently online, or a null pointer if there are none.

**accelerometer→calibrateFromPoints()**  
**accelerometer.calibrateFromPoints()****YAccelerometer**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**accelerometer→describe()accelerometer.describe()****YAccelerometer**

Returns a short text that describes unambiguously the instance of the accelerometer in the form  
TYPE ( NAME )=SERIAL . FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the accelerometer (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

**accelerometer→get\_advertisedValue()**

**YAccelerometer**

**accelerometer→advertisedValue()**

**accelerometer.get\_advertisedValue()**

---

Returns the current value of the accelerometer (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the accelerometer (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**accelerometer→get\_currentRawValue()**  
**accelerometer→currentRawValue()**  
**accelerometer.get\_currentRawValue()**

---

**YAccelerometer**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**accelerometer→get\_currentValue()**  
**accelerometer→currentValue()**  
**accelerometer.get\_currentValue()**

**YAccelerometer**

Returns the current value of the acceleration.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the acceleration

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

**accelerometer→get\_errorMessage()**  
**accelerometer→errorMessage()**  
**accelerometer.get\_errorMessage()**

---

**YAccelerometer**

Returns the error message of the latest error with the accelerometer.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the accelerometer object

**accelerometer→get\_errorType()**  
**accelerometer→errorType()**  
**accelerometer.get\_errorType()**

**YAccelerometer**

Returns the numerical error code of the latest error with the accelerometer.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the accelerometer object

**accelerometer→get\_friendlyName()**  
**accelerometer→friendlyName()**  
**accelerometer.get\_friendlyName()**

**YAccelerometer**

Returns a global identifier of the accelerometer in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the accelerometer if they are defined, otherwise the serial number of the module and the hardware identifier of the accelerometer (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the accelerometer using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

`accelerometer→get_functionDescriptor()`  
`accelerometer→functionDescriptor()`  
`accelerometer.get_functionDescriptor()`

**YAccelerometer**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

`def get_functionDescriptor( )`

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**accelerometer→get\_functionId()**  
**accelerometer→functionId()**  
**accelerometer.get\_functionId()**

---

**YAccelerometer**

Returns the hardware identifier of the accelerometer, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the accelerometer (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**accelerometer→get.hardwareId()**  
**accelerometer→hardwareId()**  
**accelerometer.get.hardwareId()**

**YAccelerometer**

Returns the unique hardware identifier of the accelerometer in the form SERIAL.FUNCTIONID.

**def get.hardwareId( )**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the accelerometer. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the accelerometer (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**accelerometer→get\_highestValue()**  
**accelerometer→highestValue()**  
**accelerometer.get\_highestValue()**

**YAccelerometer**

---

Returns the maximal value observed for the acceleration since the device was started.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the acceleration since the device was started

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**accelerometer→get\_logFrequency()**  
**accelerometer→logFrequency()**  
**accelerometer.get\_logFrequency()**

**YAccelerometer**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**accelerometer→get\_logicalName()**  
**accelerometer→logicalName()**  
**accelerometer.get\_logicalName()**

**YAccelerometer**

---

Returns the logical name of the accelerometer.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the accelerometer. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**accelerometer→get\_lowestValue()**  
**accelerometer→lowestValue()**  
**accelerometer.get\_lowestValue()**

**YAccelerometer**

Returns the minimal value observed for the acceleration since the device was started.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the acceleration since the device was started

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**accelerometer→get\_module()****YAccelerometer****accelerometer→module()accelerometer.get\_module()**

Gets the **YModule** object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of **YModule** is not shown as online.

**Returns :**

an instance of **YModule**

**accelerometer→get\_recordedData()**  
**accelerometer→recordedData()**  
**accelerometer.get\_recordedData()**

**YAccelerometer**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**accelerometer→get\_reportFrequency()**  
**accelerometer→reportFrequency()**  
**accelerometer.get\_reportFrequency()**

YAccelerometer

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**accelerometer→get\_resolution()**  
**accelerometer→resolution()**  
**accelerometer.get\_resolution()**

**YAccelerometer**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**accelerometer→get\_unit()**

**YAccelerometer**

**accelerometer→unit()accelerometer.get\_unit()**

---

Returns the measuring unit for the acceleration.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the acceleration

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**accelerometer→get(userData)**  
**accelerometer→userData()**  
**accelerometer.get(userData())**

**YAccelerometer**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

`def get(userData( )`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**accelerometer→get\_xValue()**

**YAccelerometer**

**accelerometer→xValue()accelerometer.get\_xValue()**

---

Returns the X component of the acceleration, as a floating point number.

```
def get_xValue( )
```

**Returns :**

a floating point number corresponding to the X component of the acceleration, as a floating point number

On failure, throws an exception or returns Y\_XVALUE\_INVALID.

**accelerometer→get\_yValue()**

**YAccelerometer**

**accelerometer→yValue()accelerometer.get\_yValue()**

---

Returns the Y component of the acceleration, as a floating point number.

```
def get_yValue( )
```

**Returns :**

a floating point number corresponding to the Y component of the acceleration, as a floating point number

On failure, throws an exception or returns `Y_YVALUE_INVALID`.

**accelerometer→get\_zValue()**

**YAccelerometer**

**accelerometer→zValue()accelerometer.get\_zValue()**

---

Returns the Z component of the acceleration, as a floating point number.

```
def get_zValue( )
```

**Returns :**

a floating point number corresponding to the Z component of the acceleration, as a floating point number

On failure, throws an exception or returns Y\_ZVALUE\_INVALID.

**accelerometer→isOnline()****YAccelerometer**

Checks if the accelerometer is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the accelerometer in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the accelerometer.

**Returns :**

`true` if the accelerometer can be reached, and `false` otherwise

**accelerometer→load()accelerometer.load()****YAccelerometer**

Preloads the accelerometer cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**accelerometer→loadCalibrationPoints()**  
**accelerometer.loadCalibrationPoints()****YAccelerometer**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**accelerometer→nextAccelerometer()**  
**accelerometer.nextAccelerometer()**

---

**YAccelerometer**

Continues the enumeration of accelerometers started using `yFirstAccelerometer()`.

```
def nextAccelerometer( )
```

**Returns :**

a pointer to a `YAccelerometer` object, corresponding to an accelerometer currently online, or a `null` pointer if there are no more accelerometers to enumerate.

**accelerometer→registerTimedReportCallback()**  
**accelerometer.registerTimedReportCallback()****YAccelerometer**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**accelerometer→registerValueCallback()  
accelerometer.registerValueCallback()****YAccelerometer**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**accelerometer→set\_highestValue()**  
**accelerometer→setHighestValue()**  
**accelerometer.set\_highestValue()**

**YAccelerometer**

Changes the recorded maximal value observed.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**accelerometer→set\_logFrequency()**  
**accelerometer→setLogFrequency()**  
**accelerometer.set\_logFrequency()**

**YAccelerometer**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

**YAPI\_SUCCESS** if the call succeeds.

On failure, throws an exception or returns a negative error code.

**accelerometer→set\_logicalName()**  
**accelerometer→setLogicalName()**  
**accelerometer.set\_logicalName()**

**YAccelerometer**

Changes the logical name of the accelerometer.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the accelerometer.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

`accelerometer→set_lowestValue()`  
`accelerometer→setLowestValue()`  
`accelerometer.set_lowestValue()`

YAccelerometer

Changes the recorded minimal value observed.

```
def set_lowestValue( newval )
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**accelerometer→set\_reportFrequency()**  
**accelerometer→setReportFrequency()**  
**accelerometer.set\_reportFrequency()**

**YAccelerometer**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**accelerometer→set\_resolution()**  
**accelerometer→setResolution()**  
**accelerometer.set\_resolution()**

**YAccelerometer**

Changes the resolution of the measured physical values.

```
def set_resolution( newval )
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**accelerometer→set(userData)**  
**accelerometer→setUserData()**  
**accelerometer.set(userData)**

**YAccelerometer**

Stores a user context provided as argument in the userData attribute of the function.

```
def setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

### 3.3. AnButton function interface

Yoctopuce application programming interface allows you to measure the state of a simple button as well as to read an analog potentiometer (variable resistance). This can be used for instance with a continuous rotating knob, a throttle grip or a joystick. The module is capable to calibrate itself on min and max values, in order to compute a calibrated value that varies proportionally with the potentiometer position, regardless of its total resistance.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_anbutton.js'></script>
nodejs var yoctolib = require('yoctolib');
var YAnButton = yoctolib.YAnButton;
php require_once('yocto_anbutton.php');
cpp #include "yocto_anbutton.h"
m #import "yocto_anbutton.h"
pas uses yocto_anbutton;
vb yocto_anbutton.vb
cs yocto_anbutton.cs
java import com.yoctopuce.YoctoAPI.YAnButton;
py from yocto_anbutton import *

```

#### Global functions

##### **yFindAnButton(func)**

Retrieves an analog input for a given identifier.

##### **yFirstAnButton()**

Starts the enumeration of analog inputs currently accessible.

#### YAnButton methods

##### **anbutton→describe()**

Returns a short text that describes unambiguously the instance of the analog input in the form TYPE (NAME )=SERIAL.FUNCTIONID.

##### **anbutton→get\_advertisedValue()**

Returns the current value of the analog input (no more than 6 characters).

##### **anbutton→get\_analogCalibration()**

Tells if a calibration process is currently ongoing.

##### **anbutton→get\_calibratedValue()**

Returns the current calibrated input value (between 0 and 1000, included).

##### **anbutton→get\_calibrationMax()**

Returns the maximal value measured during the calibration (between 0 and 4095, included).

##### **anbutton→get\_calibrationMin()**

Returns the minimal value measured during the calibration (between 0 and 4095, included).

##### **anbutton→get\_errorMessage()**

Returns the error message of the latest error with the analog input.

##### **anbutton→get\_errorType()**

Returns the numerical error code of the latest error with the analog input.

##### **anbutton→get\_friendlyName()**

Returns a global identifier of the analog input in the format MODULE\_NAME . FUNCTION\_NAME.

##### **anbutton→get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

**anbutton→get\_functionId()**

Returns the hardware identifier of the analog input, without reference to the module.

**anbutton→get\_hardwareId()**

Returns the unique hardware identifier of the analog input in the form SERIAL.FUNCTIONID.

**anbutton→get\_isPressed()**

Returns true if the input (considered as binary) is active (closed contact), and false otherwise.

**anbutton→get\_lastTimePressed()**

Returns the number of elapsed milliseconds between the module power on and the last time the input button was pressed (the input contact transitionned from open to closed).

**anbutton→get\_lastTimeReleased()**

Returns the number of elapsed milliseconds between the module power on and the last time the input button was released (the input contact transitionned from closed to open).

**anbutton→get\_logicalName()**

Returns the logical name of the analog input.

**anbutton→get\_module()**

Gets the YModule object for the device on which the function is located.

**anbutton→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**anbutton→get\_pulseCounter()**

Returns the pulse counter value

**anbutton→get\_pulseTimer()**

Returns the timer of the pulses counter (ms)

**anbutton→get\_rawValue()**

Returns the current measured input value as-is (between 0 and 4095, included).

**anbutton→get\_sensitivity()**

Returns the sensibility for the input (between 1 and 1000) for triggering user callbacks.

**anbutton→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set(userData).

**anbutton→isOnline()**

Checks if the analog input is currently reachable, without raising any error.

**anbutton→isOnline\_async(callback, context)**

Checks if the analog input is currently reachable, without raising any error (asynchronous version).

**anbutton→load(msValidity)**

Preloads the analog input cache with a specified validity duration.

**anbutton→load\_async(msValidity, callback, context)**

Preloads the analog input cache with a specified validity duration (asynchronous version).

**anbutton→nextAnButton()**

Continues the enumeration of analog inputs started using yFirstAnButton( ).

**anbutton→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**anbutton→resetCounter()**

Returns the pulse counter value as well as his timer

**anbutton→set\_analogCalibration(newval)**

Starts or stops the calibration process.

**anbutton→set\_calibrationMax(newval)**

### 3. Reference

---

Changes the maximal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration.

**anbutton→set\_calibrationMin(newval)**

Changes the minimal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration.

**anbutton→set\_logicalName(newval)**

Changes the logical name of the analog input.

**anbutton→set\_sensitivity(newval)**

Changes the sensibility for the input (between 1 and 1000) for triggering user callbacks.

**anbutton→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**anbutton→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**YAnButton.FindAnButton()****YAnButton****yFindAnButton()YAnButton.FindAnButton()**

Retrieves an analog input for a given identifier.

```
def FindAnButton( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the analog input is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAnButton.isOnline()` to test if the analog input is indeed online at a given time. In case of ambiguity when looking for an analog input by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the analog input

**Returns :**

a `YAnButton` object allowing you to drive the analog input.

## **YAnButton.FirstAnButton()**

**YAnButton**

## **yFirstAnButton()YAnButton.FirstAnButton()**

---

Starts the enumeration of analog inputs currently accessible.

```
def FirstAnButton( )
```

Use the method `YAnButton.nextAnButton()` to iterate on next analog inputs.

**Returns :**

a pointer to a `YAnButton` object, corresponding to the first analog input currently online, or a null pointer if there are none.

**anbutton→describe()anbutton.describe()****YAnButton**

Returns a short text that describes unambiguously the instance of the analog input in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the analog input (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**anbutton→get\_advertisedValue()**  
**anbutton→advertisedValue()**  
**anbutton.get\_advertisedValue()**

**YAnButton**

---

Returns the current value of the analog input (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the analog input (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**anbutton→get\_analogCalibration()**  
**anbutton→analogCalibration()**  
**anbutton.get\_analogCalibration()**

**YAnButton**

Tells if a calibration process is currently ongoing.

```
def get_analogCalibration( )
```

**Returns :**

either Y\_ANALOGCALIBRATION\_OFF or Y\_ANALOGCALIBRATION\_ON

On failure, throws an exception or returns Y\_ANALOGCALIBRATION\_INVALID.

**anbutton→get\_calibratedValue()**  
**anbutton→calibratedValue()**  
**anbutton.get\_calibratedValue()**

---

**YAnButton**

Returns the current calibrated input value (between 0 and 1000, included).

```
def get_calibratedValue( )
```

**Returns :**

an integer corresponding to the current calibrated input value (between 0 and 1000, included)

On failure, throws an exception or returns Y\_CALIBRATEDVALUE\_INVALID.

**anbutton→get\_calibrationMax()**  
**anbutton→calibrationMax()**  
**anbutton.get\_calibrationMax()**

**YAnButton**

Returns the maximal value measured during the calibration (between 0 and 4095, included).

```
def get_calibrationMax( )
```

**Returns :**

an integer corresponding to the maximal value measured during the calibration (between 0 and 4095, included)

On failure, throws an exception or returns Y\_CALIBRATIONMAX\_INVALID.

**anbutton→get\_calibrationMin()**  
**anbutton→calibrationMin()**  
**anbutton.get\_calibrationMin()**

**YAnButton**

Returns the minimal value measured during the calibration (between 0 and 4095, included).

```
def get_calibrationMin( )
```

**Returns :**

an integer corresponding to the minimal value measured during the calibration (between 0 and 4095, included)

On failure, throws an exception or returns Y\_CALIBRATIONMIN\_INVALID.

---

**anbutton→getErrorMessage()**  
**anbutton→errorMessage()**  
**anbutton.getErrorMessage()**

**YAnButton**

Returns the error message of the latest error with the analog input.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the analog input object

**anbutton→get\_errorType()**

**YAnButton**

**anbutton→errorType()anbutton.get\_errorType()**

---

Returns the numerical error code of the latest error with the analog input.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the analog input object

**anbutton→get\_friendlyName()**  
**anbutton→friendlyName()**  
**anbutton.get\_friendlyName()**

**YAnButton**

Returns a global identifier of the analog input in the format MODULE\_NAME . FUNCTION\_NAME.

**def get\_friendlyName( )**

The returned string uses the logical names of the module and of the analog input if they are defined, otherwise the serial number of the module and the hardware identifier of the analog input (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the analog input using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

---

<b>anbutton→get_functionDescriptor()</b>	<b>YAnButton</b>
<b>anbutton→functionDescriptor()</b>	
<b>anbutton.get_functionDescriptor()</b>	

---

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

---

**anbutton→get\_functionId()****YAnButton****anbutton→functionId()anbutton.get\_functionId()**

---

Returns the hardware identifier of the analog input, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the analog input (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**anbutton→get\_hardwareId()**

**YAnButton**

**anbutton→hardwareId()anbutton.get\_hardwareId()**

---

Returns the unique hardware identifier of the analog input in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the analog input. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the analog input (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

---

**anbutton→get\_isPressed()****YAnButton****anbutton→isPressed()anbutton.get\_isPressed()**

Returns true if the input (considered as binary) is active (closed contact), and false otherwise.

```
def get_isPressed( )
```

**Returns :**

either Y\_ISPRESSED\_FALSE or Y\_ISPRESSED\_TRUE, according to true if the input (considered as binary) is active (closed contact), and false otherwise

On failure, throws an exception or returns Y\_ISPRESSED\_INVALID.

**anbutton→get\_lastTimePressed()**  
**anbutton→lastTimePressed()**  
**anbutton.get\_lastTimePressed()**

**YAnButton**

Returns the number of elapsed milliseconds between the module power on and the last time the input button was pressed (the input contact transitionned from open to closed).

```
def get_lastTimePressed( )
```

**Returns :**

an integer corresponding to the number of elapsed milliseconds between the module power on and the last time the input button was pressed (the input contact transitionned from open to closed)

On failure, throws an exception or returns Y\_LASTTIMEPRESSED\_INVALID.

**anbutton→get\_lastTimeReleased()**  
**anbutton→lastTimeReleased()**  
**anbutton.get\_lastTimeReleased()**

**YAnButton**

Returns the number of elapsed milliseconds between the module power on and the last time the input button was released (the input contact transitionned from closed to open).

```
def get_lastTimeReleased( )
```

**Returns :**

an integer corresponding to the number of elapsed milliseconds between the module power on and the last time the input button was released (the input contact transitionned from closed to open)

On failure, throws an exception or returns Y\_LASTTIMERELEASED\_INVALID.

**anbutton→get\_logicalName()**

**YAnButton**

**anbutton→logicalName()anbutton.get\_logicalName()**

---

Returns the logical name of the analog input.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the analog input. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**anbutton→get\_module()****YAnButton****anbutton→module()anbutton.get\_module()**

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

**anbutton→get\_pulseCounter()**  
**anbutton→pulseCounter()**  
**anbutton.get\_pulseCounter()**

---

**YAnButton**

Returns the pulse counter value

```
def get_pulseCounter( )
```

**Returns :**

an integer corresponding to the pulse counter value

On failure, throws an exception or returns Y\_PULSECOUNTERR\_INVALID.

---

**anbutton→get\_pulseTimer()****YAnButton****anbutton→pulseTimer()anbutton.get\_pulseTimer()**

---

Returns the timer of the pulses counter (ms)

```
def get_pulseTimer( )
```

**Returns :**

an integer corresponding to the timer of the pulses counter (ms)

On failure, throws an exception or returns Y\_PULSE\_TIMER\_INVALID.

**anbutton→get\_rawValue()**

**YAnButton**

**anbutton→rawValue()anbutton.get\_rawValue()**

---

Returns the current measured input value as-is (between 0 and 4095, included).

```
def get_rawValue( )
```

**Returns :**

an integer corresponding to the current measured input value as-is (between 0 and 4095, included)

On failure, throws an exception or returns Y\_RAWVALUE\_INVALID.

**anbutton→get\_sensitivity()****YAnButton****anbutton→sensitivity()|anbutton.get\_sensitivity()**

Returns the sensibility for the input (between 1 and 1000) for triggering user callbacks.

```
def get_sensitivity( )
```

**Returns :**

an integer corresponding to the sensibility for the input (between 1 and 1000) for triggering user callbacks

On failure, throws an exception or returns Y\_SENSITIVITY\_INVALID.

**anbutton→get(userData)**

**YAnButton**

**anbutton→userData()anbutton.get(userData)**

---

Returns the value of the userData attribute, as previously stored using method set(userData).

```
def get(userData) :
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**anbutton→isOnline()|anbutton.isOnline()****YAnButton**

Checks if the analog input is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the analog input in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the analog input.

**Returns :**

`true` if the analog input can be reached, and `false` otherwise

**anbutton→load()****YAnButton**

Preloads the analog input cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**anbutton→nextAnButton()****YAnButton**

Continues the enumeration of analog inputs started using `yFirstAnButton()`.

```
def nextAnButton( )
```

**Returns :**

a pointer to a `YAnButton` object, corresponding to an analog input currently online, or a null pointer if there are no more analog inputs to enumerate.

**anbutton→registerValueCallback()  
anbutton.registerValueCallback()****YAnButton**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**anbutton→resetCounter()|anbutton.resetCounter()****YAnButton**

Returns the pulse counter value as well as his timer

```
def resetCounter( )
```

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**anbutton→set\_analogCalibration()**  
**anbutton→setAnalogCalibration()**  
**anbutton.set\_analogCalibration()**

**YAnButton**

Starts or stops the calibration process.

```
def set_analogCalibration( newval)
```

Remember to call the `saveToFlash()` method of the module at the end of the calibration if the modification must be kept.

**Parameters :**

**newval** either `Y_ANALOGCALIBRATION_OFF` or `Y_ANALOGCALIBRATION_ON`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**anbutton→set\_calibrationMax()**  
**anbutton→setCalibrationMax()**  
**anbutton.set\_calibrationMax()**

**YAnButton**

Changes the maximal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration.

```
def set_calibrationMax( newval )
```

Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

**Parameters :**

**newval** an integer corresponding to the maximal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**anbutton→set\_calibrationMin()**  
**anbutton→setCalibrationMin()**  
**anbutton.set\_calibrationMin()**

**YAnButton**

Changes the minimal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration.

```
def set_calibrationMin( newval)
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** an integer corresponding to the minimal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**anbutton→set\_logicalName()**  
**anbutton→setLogicalName()**  
**anbutton.set\_logicalName()**

**YAnButton**

Changes the logical name of the analog input.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the analog input.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**anbutton→set\_sensitivity()****YAnButton****anbutton→setSensitivity()anbutton.set\_sensitivity()**

Changes the sensibility for the input (between 1 and 1000) for triggering user callbacks.

```
def set_sensitivity( newval)
```

The sensibility is used to filter variations around a fixed value, but does not preclude the transmission of events when the input value evolves constantly in the same direction. Special case: when the value 1000 is used, the callback will only be thrown when the logical state of the input switches from pressed to released and back. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** an integer corresponding to the sensibility for the input (between 1 and 1000) for triggering user callbacks

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**anbutton→set(userData)****YAnButton****anbutton→setUserData()anbutton.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData) data
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.4. CarbonDioxide function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_carbondioxide.js'></script>
nodejs var yoctolib = require('yoctolib');
var YCarbonDioxide = yoctolib.YCarbonDioxide;
php require_once('yocto_carbondioxide.php');
cpp #include "yocto_carbondioxide.h"
m #import "yocto_carbondioxide.h"
pas uses yocto_carbondioxide;
vb yocto_carbondioxide.vb
cs yocto_carbondioxide.cs
java import com.yoctopuce.YoctoAPI.YCarbonDioxide;
py from yocto_carbondioxide import *

```

### Global functions

#### **yFindCarbonDioxide(func)**

Retrieves a CO2 sensor for a given identifier.

#### **yFirstCarbonDioxide()**

Starts the enumeration of CO2 sensors currently accessible.

### YCarbonDioxide methods

#### **carbondioxide→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### **carbondioxide→describe()**

Returns a short text that describes unambiguously the instance of the CO2 sensor in the form TYPE (NAME) = SERIAL . FUNCTIONID.

#### **carbondioxide→get\_advertisedValue()**

Returns the current value of the CO2 sensor (no more than 6 characters).

#### **carbondioxide→get\_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### **carbondioxide→get\_currentValue()**

Returns the current value of the CO2 concentration.

#### **carbondioxide→get\_errorMessage()**

Returns the error message of the latest error with the CO2 sensor.

#### **carbondioxide→get\_errorType()**

Returns the numerical error code of the latest error with the CO2 sensor.

#### **carbondioxide→get\_friendlyName()**

Returns a global identifier of the CO2 sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### **carbondioxide→get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### **carbondioxide→get\_functionId()**

Returns the hardware identifier of the CO2 sensor, without reference to the module.

#### **carbondioxide→get\_hardwareId()**

Returns the unique hardware identifier of the CO2 sensor in the form SERIAL . FUNCTIONID.

**carbondioxide→get\_highestValue()**

Returns the maximal value observed for the CO2 concentration since the device was started.

**carbondioxide→get\_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**carbondioxide→get\_logicalName()**

Returns the logical name of the CO2 sensor.

**carbondioxide→get\_lowestValue()**

Returns the minimal value observed for the CO2 concentration since the device was started.

**carbondioxide→get\_module()**

Gets the YModule object for the device on which the function is located.

**carbondioxide→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**carbondioxide→get\_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**carbondioxide→get\_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**carbondioxide→get\_resolution()**

Returns the resolution of the measured values.

**carbondioxide→get\_unit()**

Returns the measuring unit for the CO2 concentration.

**carbondioxide→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set(userData).

**carbondioxide→isOnline()**

Checks if the CO2 sensor is currently reachable, without raising any error.

**carbondioxide→isOnline\_async(callback, context)**

Checks if the CO2 sensor is currently reachable, without raising any error (asynchronous version).

**carbondioxide→load(msValidity)**

Preloads the CO2 sensor cache with a specified validity duration.

**carbondioxide→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

**carbondioxide→load\_async(msValidity, callback, context)**

Preloads the CO2 sensor cache with a specified validity duration (asynchronous version).

**carbondioxide→nextCarbonDioxide()**

Continues the enumeration of CO2 sensors started using yFirstCarbonDioxide( ).

**carbondioxide→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**carbondioxide→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**carbondioxide→set\_highestValue(newval)**

Changes the recorded maximal value observed.

**carbondioxide→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**carbondioxide→set\_logicalName(newval)**

Changes the logical name of the CO2 sensor.

### 3. Reference

---

**carbondioxide→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**carbondioxide→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**carbondioxide→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**carbondioxide→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**carbondioxide→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YCarbonDioxide.FindCarbonDioxide() yFindCarbonDioxide() YCarbonDioxide.FindCarbonDioxide()

## YCarbonDioxide

Retrieves a CO2 sensor for a given identifier.

```
def FindCarbonDioxide( func )
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the CO2 sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCarbonDioxide.isOnline()` to test if the CO2 sensor is indeed online at a given time. In case of ambiguity when looking for a CO2 sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the CO2 sensor

### Returns :

a `YCarbonDioxide` object allowing you to drive the CO2 sensor.

## **YCarbonDioxide.FirstCarbonDioxide()**

## **YCarbonDioxide**

### **yFirstCarbonDioxide()**

### **YCarbonDioxide.FirstCarbonDioxide()**

---

Starts the enumeration of CO2 sensors currently accessible.

```
def FirstCarbonDioxide( )
```

Use the method `YCarbonDioxide.nextCarbonDioxide()` to iterate on next CO2 sensors.

#### **Returns :**

a pointer to a `YCarbonDioxide` object, corresponding to the first CO2 sensor currently online, or a null pointer if there are none.

**carbondioxide→calibrateFromPoints()  
carbondioxide.calibrateFromPoints()****YCarbonDioxide**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**carbondioxide→describe()carbon dioxide.describe()****YCarbonDioxide**

Returns a short text that describes unambiguously the instance of the CO2 sensor in the form  
TYPE ( NAME )=SERIAL . FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the CO2 sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

**carbondioxide→get\_advertisedValue()**  
**carbondioxide→advertisedValue()**  
**carbondioxide.get\_advertisedValue()**

**YCarbonDioxide**

Returns the current value of the CO2 sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the CO2 sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**carbondioxide→get\_currentRawValue()**  
**carbondioxide→currentRawValue()**  
**carbondioxide.get\_currentRawValue()**

**YCarbonDioxide**

---

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**carbondioxide→get\_currentValue()**  
**carbondioxide→currentValue()**  
**carbondioxide.get\_currentValue()**

**YCarbonDioxide**

Returns the current value of the CO2 concentration.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the CO2 concentration

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

**carbondioxide→get\_errorMessage()**  
**carbondioxide→errorMessage()**  
**carbondioxide.get\_errorMessage()**

**YCarbonDioxide**

---

Returns the error message of the latest error with the CO2 sensor.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the CO2 sensor object

**carbondioxide→get\_errorType()**  
**carbondioxide→errorType()**  
**carbondioxide.get\_errorType()**

**YCarbonDioxide**

---

Returns the numerical error code of the latest error with the CO2 sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the CO2 sensor object

**carbondioxide→get\_friendlyName()**  
**carbondioxide→friendlyName()**  
**carbondioxide.get\_friendlyName()**

**YCarbonDioxide**

Returns a global identifier of the CO2 sensor in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the CO2 sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the CO2 sensor (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the CO2 sensor using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**carbondioxide→get\_functionDescriptor()**  
**carbondioxide→functionDescriptor()**  
**carbondioxide.get\_functionDescriptor()**

**YCarbonDioxide**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**carbondioxide→get\_functionId()**  
**carbondioxide→functionId()**  
**carbondioxide.get\_functionId()**

**YCarbonDioxide**

---

Returns the hardware identifier of the CO2 sensor, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the CO2 sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**carbondioxide→get.hardwareId()**  
**carbondioxide→hardwareId()**  
**carbondioxide.get.hardwareId()**

**YCarbonDioxide**

Returns the unique hardware identifier of the CO2 sensor in the form SERIAL.FUNCTIONID.

**def get.hardwareId( )**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the CO2 sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the CO2 sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**carbondioxide→get\_highestValue()**  
**carbondioxide→highestValue()**  
**carbondioxide.get\_highestValue()**

**YCarbonDioxide**

Returns the maximal value observed for the CO2 concentration since the device was started.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the CO2 concentration since the device was started

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**carbondioxide→get\_logFrequency()**  
**carbondioxide→logFrequency()**  
**carbondioxide.get\_logFrequency()**

**YCarbonDioxide**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**carbondioxide→get\_logicalName()**  
**carbondioxide→logicalName()**  
**carbondioxide.get\_logicalName()**

**YCarbonDioxide**

---

Returns the logical name of the CO2 sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the CO2 sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**carbondioxide→get\_lowestValue()**  
**carbondioxide→lowestValue()**  
**carbondioxide.get\_lowestValue()**

**YCarbonDioxide**

Returns the minimal value observed for the CO<sub>2</sub> concentration since the device was started.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the CO<sub>2</sub> concentration since the device was started

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**carbondioxide→get\_module()**  
**carbondioxide→module()**  
**carbondioxide.get\_module()**

---

**YCarbonDioxide**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**carbondioxide→get\_recordedData()**  
**carbondioxide→recordedData()**  
**carbondioxide.get\_recordedData()**

**YCarbonDioxide**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**carbondioxide→get\_reportFrequency()**

**YCarbonDioxide**

**carbondioxide→reportFrequency()**

**carbondioxide.get\_reportFrequency()**

---

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**carbondioxide→get\_resolution()**  
**carbondioxide→resolution()**  
**carbondioxide.get\_resolution()**

**YCarbonDioxide**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**carbondioxide→get\_unit()**

**YCarbonDioxide**

**carbondioxide→unit()carbondioxide.get\_unit()**

---

Returns the measuring unit for the CO2 concentration.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the CO2 concentration

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**carbondioxide→get(userData)**  
**carbondioxide→userData()**  
**carbondioxide.get(userData)**

**YCarbonDioxide**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

## carbondioxide→isOnline()carbon dioxide.isOnline()

## YCarbonDioxide

---

Checks if the CO2 sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the CO2 sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the CO2 sensor.

**Returns :**

true if the CO2 sensor can be reached, and false otherwise

**carbondioxide→load()carbon dioxide.load()****YCarbonDioxide**

Preloads the CO2 sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**carbon dioxide → loadCalibrationPoints()  
carbon dioxide.loadCalibrationPoints()****YCarbonDioxide**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**carbondioxide→nextCarbonDioxide()**  
**carbondioxide.nextCarbonDioxide()****YCarbonDioxide**

Continues the enumeration of CO2 sensors started using `yFirstCarbonDioxide()`.

```
def nextCarbonDioxide( )
```

**Returns :**

a pointer to a `YCarbonDioxide` object, corresponding to a CO2 sensor currently online, or a `null` pointer if there are no more CO2 sensors to enumerate.

**carbon dioxide → registerTimedReportCallback()**  
**carbon dioxide.registerTimedReportCallback()****YCarbonDioxide**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**carbondioxide→registerValueCallback()**  
**carbondioxide.registerValueCallback()****YCarbonDioxide**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**carbondioxide→set\_highestValue()**  
**carbondioxide→setHighestValue()**  
**carbondioxide.set\_highestValue()**

**YCarbonDioxide**

---

Changes the recorded maximal value observed.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**carbondioxide→set\_logFrequency()**  
**carbondioxide→setLogFrequency()**  
**carbondioxide.set\_logFrequency()**

**YCarbonDioxide**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**carbondioxide→set\_logicalName()**  
**carbondioxide→setLogicalName()**  
**carbondioxide.set\_logicalName()**

**YCarbonDioxide**

Changes the logical name of the CO2 sensor.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the CO2 sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**carbondioxide→set\_lowestValue()**  
**carbondioxide→setLowestValue()**  
**carbondioxide.set\_lowestValue()**

**YCarbonDioxide**

Changes the recorded minimal value observed.

```
def set_lowestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**carbon dioxide → set\_reportFrequency()**  
**carbon dioxide → setReportFrequency()**  
**carbon dioxide.set\_reportFrequency()****YCarbonDioxide**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**carbondioxide→set\_resolution()**  
**carbondioxide→setResolution()**  
**carbondioxide.set\_resolution()**

**YCarbonDioxide**

Changes the resolution of the measured physical values.

```
def set_resolution( newval )
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**carbondioxide→set(userData)**  
**carbondioxide→setUserData()**  
**carbondioxide.set(userData)**

---

**YCarbonDioxide**

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.5. ColorLed function interface

Yoctopuce application programming interface allows you to drive a color led using RGB coordinates as well as HSL coordinates. The module performs all conversions from RGB to HSL automatically. It is then self-evident to turn on a led with a given hue and to progressively vary its saturation or lightness. If needed, you can find more information on the difference between RGB and HSL in the section following this one.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_colorled.js'></script>
node.js	var yoctolib = require('yoctolib');
php	var YColorLed = yoctolib.YColorLed;
cpp	require_once('yocto_colorled.php');
m	#include "yocto_colorled.h"
pas	#import "yocto_colorled.h"
vb	uses yocto_colorled;
cs	yocto_colorled.vb
java	yocto_colorled.cs
py	import com.yoctopuce.YoctoAPI.YColorLed;
	from yocto_colorled import *

### Global functions

#### yFindColorLed(func)

Retrieves an RGB led for a given identifier.

#### yFirstColorLed()

Starts the enumeration of RGB leds currently accessible.

### YColorLed methods

#### colorled→describe()

Returns a short text that describes unambiguously the instance of the RGB led in the form TYPE (NAME) = SERIAL . FUNCTIONID.

#### colorled→get\_advertisedValue()

Returns the current value of the RGB led (no more than 6 characters).

#### colorled→get\_errorMessage()

Returns the error message of the latest error with the RGB led.

#### colorled→get\_errorType()

Returns the numerical error code of the latest error with the RGB led.

#### colorled→get\_friendlyName()

Returns a global identifier of the RGB led in the format MODULE\_NAME . FUNCTION\_NAME.

#### colorled→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### colorled→get\_functionId()

Returns the hardware identifier of the RGB led, without reference to the module.

#### colorled→get\_hardwareId()

Returns the unique hardware identifier of the RGB led in the form SERIAL . FUNCTIONID.

#### colorled→get\_hslColor()

Returns the current HSL color of the led.

#### colorled→get\_logicalName()

Returns the logical name of the RGB led.

### 3. Reference

#### **colorled→get\_module()**

Gets the YModule object for the device on which the function is located.

#### **colorled→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

#### **colorled→get\_rgbColor()**

Returns the current RGB color of the led.

#### **colorled→get\_rgbColorAtPowerOn()**

Returns the configured color to be displayed when the module is turned on.

#### **colorled→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set(userData).

#### **colorled→hslMove(hsl\_target, ms\_duration)**

Performs a smooth transition in the HSL color space between the current color and a target color.

#### **colorled→isOnline()**

Checks if the RGB led is currently reachable, without raising any error.

#### **colorled→isOnline\_async(callback, context)**

Checks if the RGB led is currently reachable, without raising any error (asynchronous version).

#### **colorled→load(msValidity)**

Preloads the RGB led cache with a specified validity duration.

#### **colorled→load\_async(msValidity, callback, context)**

Preloads the RGB led cache with a specified validity duration (asynchronous version).

#### **colorled→nextColorLed()**

Continues the enumeration of RGB leds started using yFirstColorLed( ).

#### **colorled→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

#### **colorled→rgbMove(rgb\_target, ms\_duration)**

Performs a smooth transition in the RGB color space between the current color and a target color.

#### **colorled→set\_hslColor(newval)**

Changes the current color of the led, using a color HSL.

#### **colorled→set\_logicalName(newval)**

Changes the logical name of the RGB led.

#### **colorled→set\_rgbColor(newval)**

Changes the current color of the led, using a RGB color.

#### **colorled→set\_rgbColorAtPowerOn(newval)**

Changes the color that the led will display by default when the module is turned on.

#### **colorled→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

#### **colorled→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YColorLed.FindColorLed() yFindColorLed()YColorLed.FindColorLed()

YColorLed

Retrieves an RGB led for a given identifier.

```
def FindColorLed( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the RGB led is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YColorLed.isOnline()` to test if the RGB led is indeed online at a given time. In case of ambiguity when looking for an RGB led by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the RGB led

### Returns :

a `YColorLed` object allowing you to drive the RGB led.

## **YColorLed.FirstColorLed()**

**YColorLed**

## **yFirstColorLed()YColorLed.FirstColorLed()**

---

Starts the enumeration of RGB leds currently accessible.

```
def FirstColorLed( )
```

Use the method `YColorLed.nextColorLed()` to iterate on next RGB leds.

**Returns :**

a pointer to a `YColorLed` object, corresponding to the first RGB led currently online, or a `null` pointer if there are none.

**colorled→describe()colorled.describe()****YColorLed**

Returns a short text that describes unambiguously the instance of the RGB led in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the RGB led (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**colorled→get\_advertisedValue()**  
**colorled→advertisedValue()**  
**colorled.get\_advertisedValue()**

**YColorLed**

Returns the current value of the RGB led (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the RGB led (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**colorled→get\_errorMessage()**  
**colorled→errorMessage()**  
**colorled.get\_errorMessage()**

**YColorLed**

Returns the error message of the latest error with the RGB led.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the RGB led object

**colorled→get\_errorType()**

**YColorLed**

**colorled→errorType()colorled.get\_errorType()**

---

Returns the numerical error code of the latest error with the RGB led.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the RGB led object

**colorled→get\_friendlyName()****YColorLed****colorled→friendlyName()colorled.get\_friendlyName()**

Returns a global identifier of the RGB led in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the RGB led if they are defined, otherwise the serial number of the module and the hardware identifier of the RGB led (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the RGB led using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**colorled→get\_functionDescriptor()  
colorled→functionDescriptor()  
colorled.get\_functionDescriptor()****YColorLed**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**colorled→get\_functionId()****YColorLed****colorled→functionId()colorled.get\_functionId()**

Returns the hardware identifier of the RGB led, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the RGB led (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**colorled→get\_hardwareId()**

**YColorLed**

**colorled→hardwareId()colorled.get\_hardwareId()**

---

Returns the unique hardware identifier of the RGB led in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the RGB led. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the RGB led (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

---

**colorled→get\_hslColor()****colorled→hslColor()colorled.get\_hslColor()**

---

**YColorLed**

Returns the current HSL color of the led.

```
def get_hslColor( )
```

**Returns :**

an integer corresponding to the current HSL color of the led

On failure, throws an exception or returns Y\_HSLCOLOR\_INVALID.

**colorled→get\_logicalName()**

**YColorLed**

**colorled→logicalName()colorled.get\_logicalName()**

---

Returns the logical name of the RGB led.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the RGB led. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**colorled→get\_module()****YColorLed****colorled→module()colorled.get\_module()**

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

**colorled→get\_rgbColor()**

**YColorLed**

**colorled→rgbColor()colorled.get\_rgbColor()**

---

Returns the current RGB color of the led.

```
def get_rgbColor( )
```

**Returns :**

an integer corresponding to the current RGB color of the led

On failure, throws an exception or returns Y\_RGBCOLOR\_INVALID.

**colorled→get\_rgbColorAtPowerOn()****YColorLed****colorled→rgbColorAtPowerOn()****colorled.get\_rgbColorAtPowerOn()**

Returns the configured color to be displayed when the module is turned on.

```
def get_rgbColorAtPowerOn( )
```

**Returns :**

an integer corresponding to the configured color to be displayed when the module is turned on

On failure, throws an exception or returns Y\_RGBCOLORATPOWERON\_INVALID.

**colorled→get(userData)**

**YColorLed**

**colorled→userData()colorled.get(userData)**

---

Returns the value of the userData attribute, as previously stored using method set(userData).

```
def get(userData) :
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**colorled→hsIMove()colorled.hsIMove()****YColorLed**

Performs a smooth transition in the HSL color space between the current color and a target color.

```
def hsIMove( hsl_target, ms_duration)
```

**Parameters :**

**hsl\_target** desired HSL color at the end of the transition

**ms\_duration** duration of the transition, in millisecond

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**colorled→isOnline()colorled.isOnline()****YColorLed**

Checks if the RGB led is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the RGB led in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the RGB led.

**Returns :**

true if the RGB led can be reached, and false otherwise

**colorled→load()colorled.load()****YColorLed**

Preloads the RGB led cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## colorled→nextColorLed()colorled.nextColorLed()

YColorLed

Continues the enumeration of RGB leds started using `yFirstColorLed()`.

```
def nextColorLed( )
```

**Returns :**

a pointer to a `YColorLed` object, corresponding to an RGB led currently online, or a null pointer if there are no more RGB leds to enumerate.

**colorled→registerValueCallback()**  
**colorled.registerValueCallback()****YColorLed**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**colorled→rgbMove()colorled.rgbMove()****YColorLed**

Performs a smooth transition in the RGB color space between the current color and a target color.

```
def rgbMove( rgb_target, ms_duration)
```

**Parameters :**

**rgb\_target** desired RGB color at the end of the transition

**ms\_duration** duration of the transition, in millisecond

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**colorled→set\_hslColor()****YColorLed****colorled→setHslColor()colorled.set\_hslColor()**

Changes the current color of the led, using a color HSL.

```
def set_hslColor( newval )
```

Encoding is done as follows: 0xHHSSLL.

**Parameters :**

**newval** an integer corresponding to the current color of the led, using a color HSL

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**colorled→set\_logicalName()**  
**colorled→setLogicalName()**  
**colorled.set\_logicalName()**

**YColorLed**

Changes the logical name of the RGB led.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the RGB led.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**colorled→set\_rgbColor()****YColorLed****colorled→setRgbColor()colorled.set\_rgbColor()**

Changes the current color of the led, using a RGB color.

```
def set_rgbColor( newval )
```

Encoding is done as follows: 0xRRGGBB.

**Parameters :**

**newval** an integer corresponding to the current color of the led, using a RGB color

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**colorled→set\_rgbColorAtPowerOn()**  
**colorled→setRgbColorAtPowerOn()**  
**colorled.set\_rgbColorAtPowerOn()**

**YColorLed**

Changes the color that the led will display by default when the module is turned on.

```
def set_rgbColorAtPowerOn( newval)
```

This color will be displayed as soon as the module is powered on. Remember to call the saveToFlash( ) method of the module if the change should be kept.

**Parameters :**

**newval** an integer corresponding to the color that the led will display by default when the module is turned on

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**colorled→set(userData)****YColorLed****colorled→setUserData()colorled.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.6. Compass function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_compass.js'></script>
nodejs var yoctolib = require('yoctolib');
var YCompass = yoctolib.YCompass;
php require_once('yocto_compass.php');
cpp #include "yocto_compass.h"
m #import "yocto_compass.h"
pas uses yocto_compass;
vb yocto_compass.vb
cs yocto_compass.cs
java import com.yoctopuce.YoctoAPI.YCompass;
py from yocto_compass import *

```

### Global functions

#### **yFindCompass(func)**

Retrieves a compass for a given identifier.

#### **yFirstCompass()**

Starts the enumeration of compasses currently accessible.

### YCompass methods

#### **compass→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### **compass→describe()**

Returns a short text that describes unambiguously the instance of the compass in the form TYPE(NAME)=SERIAL.FUNCTIONID.

#### **compass→get\_advertisedValue()**

Returns the current value of the compass (no more than 6 characters).

#### **compass→get\_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### **compass→get\_currentValue()**

Returns the current value of the relative bearing.

#### **compass→get\_errorMessage()**

Returns the error message of the latest error with the compass.

#### **compass→get\_errorType()**

Returns the numerical error code of the latest error with the compass.

#### **compass→get\_friendlyName()**

Returns a global identifier of the compass in the format MODULE\_NAME.FUNCTION\_NAME.

#### **compass→get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### **compass→get\_functionId()**

Returns the hardware identifier of the compass, without reference to the module.

#### **compass→get\_hardwareId()**

Returns the unique hardware identifier of the compass in the form SERIAL.FUNCTIONID.

**compass→get\_highestValue()**

Returns the maximal value observed for the relative bearing since the device was started.

**compass→get\_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**compass→get\_logicalName()**

Returns the logical name of the compass.

**compass→get\_lowestValue()**

Returns the minimal value observed for the relative bearing since the device was started.

**compass→get\_magneticHeading()**

Returns the magnetic heading, regardless of the configured bearing.

**compass→get\_module()**

Gets the YModule object for the device on which the function is located.

**compass→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**compass→get\_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**compass→get\_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**compass→get\_resolution()**

Returns the resolution of the measured values.

**compass→get\_unit()**

Returns the measuring unit for the relative bearing.

**compass→get(userData)**

Returns the value of the userData attribute, as previously stored using method set(userData).

**compass→isOnline()**

Checks if the compass is currently reachable, without raising any error.

**compass→isOnline\_async(callback, context)**

Checks if the compass is currently reachable, without raising any error (asynchronous version).

**compass→load(msValidity)**

Preloads the compass cache with a specified validity duration.

**compass→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

**compass→load\_async(msValidity, callback, context)**

Preloads the compass cache with a specified validity duration (asynchronous version).

**compass→nextCompass()**

Continues the enumeration of compasses started using yFirstCompass( ).

**compass→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**compass→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**compass→set\_highestValue(newval)**

Changes the recorded maximal value observed.

**compass→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

### **3. Reference**

---

**compass→set\_logicalName(newval)**

Changes the logical name of the compass.

**compass→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**compass→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**compass→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**compass→set(userData)**

Stores a user context provided as argument in the userData attribute of the function.

**compass→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YCompass.FindCompass() yFindCompass()YCompass.FindCompass()

YCompass

Retrieves a compass for a given identifier.

```
def FindCompass( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the compass is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCompass.isOnline()` to test if the compass is indeed online at a given time. In case of ambiguity when looking for a compass by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the compass

**Returns :**

a `YCompass` object allowing you to drive the compass.

## **YCompass.FirstCompass()**

**YCompass**

## **yFirstCompass()YCompass.FirstCompass()**

---

Starts the enumeration of compasses currently accessible.

```
def FirstCompass( )
```

Use the method `YCompass .nextCompass( )` to iterate on next compasses.

**Returns :**

a pointer to a `YCompass` object, corresponding to the first compass currently online, or a `null` pointer if there are none.

**compass→calibrateFromPoints()**  
**compass.calibrateFromPoints()****YCompass**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**compass→describe()compass.describe()****YCompass**

Returns a short text that describes unambiguously the instance of the compass in the form  
TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the compass (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

**compass→get\_advertisedValue()**  
**compass→advertisedValue()**  
**compass.get\_advertisedValue()**

**YCompass**

Returns the current value of the compass (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the compass (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**compass→get\_currentRawValue()**  
**compass→currentRawValue()**  
**compass.get\_currentRawValue()**

---

**YCompass**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**compass→get\_currentValue()**  
**compass→currentValue()**  
**compass.get\_currentValue()**

**YCompass**

Returns the current value of the relative bearing.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the relative bearing

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

**compass→get\_errorMessage()**  
**compass→errorMessage()**  
**compass.get\_errorMessage()**

---

**YCompass**

Returns the error message of the latest error with the compass.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the compass object

---

**compass→get\_errorType()****YCompass****compass→errorType()compass.get\_errorType()**

---

Returns the numerical error code of the latest error with the compass.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the compass object

**compass→get\_friendlyName()**  
**compass→friendlyName()**  
**compass.get\_friendlyName()**

**YCompass**

Returns a global identifier of the compass in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the compass if they are defined, otherwise the serial number of the module and the hardware identifier of the compass (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the compass using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**compass→get\_functionDescriptor()**  
**compass→functionDescriptor()**  
**compass.get\_functionDescriptor()**

**YCompass**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**compass→get\_functionId()**

**YCompass**

**compass→functionId()compass.get\_functionId()**

---

Returns the hardware identifier of the compass, without reference to the module.

```
def get_functionId( )
```

For example relay1

**Returns :**

a string that identifies the compass (ex: relay1) On failure, throws an exception or returns Y\_FUNCTIONID\_INVALID.

**compass→get\_hardwareId()****YCompass****compass→hardwareId()compass.get\_hardwareId()**

Returns the unique hardware identifier of the compass in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the compass. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the compass (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**compass→get\_highestValue()**  
**compass→highestValue()**  
**compass.get\_highestValue()**

**YCompass**

Returns the maximal value observed for the relative bearing since the device was started.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the relative bearing since the device was started

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**compass→get\_logFrequency()**  
**compass→logFrequency()**  
**compass.get\_logFrequency()**

**YCompass**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**compass→get\_logicalName()**

**YCompass**

**compass→logicalName()compass.get\_logicalName()**

---

Returns the logical name of the compass.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the compass. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

---

**compass→get\_lowestValue()****YCompass****compass→lowestValue()compass.get\_lowestValue()**

---

Returns the minimal value observed for the relative bearing since the device was started.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the relative bearing since the device was started

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**compass→get\_magneticHeading()**  
**compass→magneticHeading()**  
**compass.get\_magneticHeading()**

**YCompass**

---

Returns the magnetic heading, regardless of the configured bearing.

```
def get_magneticHeading( )
```

**Returns :**

a floating point number corresponding to the magnetic heading, regardless of the configured bearing

On failure, throws an exception or returns Y\_MAGNETICHEADING\_INVALID.

---

**compass→get\_module()****YCompass****compass→module()compass.get\_module()**

---

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

**compass→get\_recordedData()**  
**compass→recordedData()**  
**compass.get\_recordedData()**

**YCompass**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime )
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**compass→get\_reportFrequency()**  
**compass→reportFrequency()**  
**compass.get\_reportFrequency()**

**YCompass**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**compass→get\_resolution()**

**YCompass**

**compass→resolution()compass.get\_resolution()**

---

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**compass→get\_unit()****YCompass****compass→unit()compass.get\_unit()**

Returns the measuring unit for the relative bearing.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the relative bearing

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**compass→get(userData)**

**YCompass**

**compass→userData()compass.get(userData)**

---

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**compass→isOnline()compass.isOnline()****YCompass**

Checks if the compass is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the compass in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the compass.

**Returns :**

`true` if the compass can be reached, and `false` otherwise

**compass→load()compass.load()****YCompass**

Preloads the compass cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**compass→loadCalibrationPoints()**  
**compass.loadCalibrationPoints()****YCompass**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## compass→nextCompass()compass.nextCompass()

## YCompass

Continues the enumeration of compasses started using `yFirstCompass()`.

```
def nextCompass( )
```

**Returns :**

a pointer to a `YCompass` object, corresponding to a compass currently online, or a `null` pointer if there are no more compasses to enumerate.

**compass→registerTimedReportCallback()  
compass.registerTimedReportCallback()****YCompass**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**compass→registerValueCallback()  
compass.registerValueCallback()****YCompass**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**compass→set\_highestValue()**  
**compass→setHighestValue()**  
**compass.set\_highestValue()**

**YCompass**

Changes the recorded maximal value observed.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**compass→set\_logFrequency()**  
**compass→setLogFrequency()**  
**compass.set\_logFrequency()**

**YCompass**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**compass→set\_logicalName()**  
**compass→setLogicalName()**  
**compass.set\_logicalName()**

**YCompass**

Changes the logical name of the compass.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the compass.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**compass→set\_lowestValue()**  
**compass→setLowestValue()**  
**compass.set\_lowestValue()**

**YCompass**

Changes the recorded minimal value observed.

```
def set_lowestValue( newval )
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**compass→set\_reportFrequency()**  
**compass→setReportFrequency()**  
**compass.set\_reportFrequency()**

**YCompass**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## compass→set\_resolution()

**YCompass**

## compass→setResolution()compass.set\_resolution()

---

Changes the resolution of the measured physical values.

```
def set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**compass→set(userData)****YCompass****compass→setUserData()compass.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.7. Current function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_current.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YCurrent = yoctolib.YCurrent;
php	require_once('yocto_current.php');
cpp	#include "yocto_current.h"
m	#import "yocto_current.h"
pas	uses yocto_current;
vb	yocto_current.vb
cs	yocto_current.cs
java	import com.yoctopuce.YoctoAPI.YCurrent;
py	from yocto_current import *

### Global functions

#### yFindCurrent(func)

Retrieves a current sensor for a given identifier.

#### yFirstCurrent()

Starts the enumeration of current sensors currently accessible.

### YCurrent methods

#### current→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### current→describe()

Returns a short text that describes unambiguously the instance of the current sensor in the form TYPE (NAME) = SERIAL . FUNCTIONID.

#### current→get\_advertisedValue()

Returns the current value of the current sensor (no more than 6 characters).

#### current→get\_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### current→get\_currentValue()

Returns the current measure for the current.

#### current→get\_errorMessage()

Returns the error message of the latest error with the current sensor.

#### current→get\_errorType()

Returns the numerical error code of the latest error with the current sensor.

#### current→get\_friendlyName()

Returns a global identifier of the current sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### current→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### current→get\_functionId()

Returns the hardware identifier of the current sensor, without reference to the module.

#### current→get\_hardwareId()

Returns the unique hardware identifier of the current sensor in the form SERIAL . FUNCTIONID.

**current→get\_highestValue()**

Returns the maximal value observed for the current.

**current→get\_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**current→get\_logicalName()**

Returns the logical name of the current sensor.

**current→get\_lowestValue()**

Returns the minimal value observed for the current.

**current→get\_module()**

Gets the YModule object for the device on which the function is located.

**current→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**current→get\_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**current→get\_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**current→get\_resolution()**

Returns the resolution of the measured values.

**current→get\_unit()**

Returns the measuring unit for the current.

**current→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set(userData).

**current→isOnline()**

Checks if the current sensor is currently reachable, without raising any error.

**current→isOnline\_async(callback, context)**

Checks if the current sensor is currently reachable, without raising any error (asynchronous version).

**current→load(msValidity)**

Preloads the current sensor cache with a specified validity duration.

**current→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

**current→load\_async(msValidity, callback, context)**

Preloads the current sensor cache with a specified validity duration (asynchronous version).

**current→nextCurrent()**

Continues the enumeration of current sensors started using yFirstCurrent( ).

**current→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**current→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**current→set\_highestValue(newval)**

Changes the recorded maximal value observed pour the current.

**current→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**current→set\_logicalName(newval)**

Changes the logical name of the current sensor.

### 3. Reference

---

**current→set\_lowestValue(newval)**

Changes the recorded minimal value observed pour the current.

**current→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**current→set\_resolution(newval)**

Changes the resolution of the measured values.

**current→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**current→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**YCurrent.FindCurrent()****YCurrent****yFindCurrent() YCurrent.FindCurrent()**

Retrieves a current sensor for a given identifier.

```
def FindCurrent( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the current sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCurrent.isOnline()` to test if the current sensor is indeed online at a given time. In case of ambiguity when looking for a current sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the current sensor

**Returns :**

a `YCurrent` object allowing you to drive the current sensor.

## **YCurrent.FirstCurrent()**

**YCurrent**

### **yFirstCurrent()YCurrent.FirstCurrent()**

Starts the enumeration of current sensors currently accessible.

```
def FirstCurrent( )
```

Use the method `YCurrent.nextCurrent( )` to iterate on next current sensors.

**Returns :**

a pointer to a `YCurrent` object, corresponding to the first current sensor currently online, or a null pointer if there are none.

**current→calibrateFromPoints()  
current.calibrateFromPoints()****YCurrent**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**current→describe()current.describe()****YCurrent**

Returns a short text that describes unambiguously the instance of the current sensor in the form  
TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the current sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

**current→get\_advertisedValue()**  
**current→advertisedValue()**  
**current.get\_advertisedValue()**

**YCurrent**

Returns the current value of the current sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the current sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**current→get\_currentRawValue()**  
**current→currentRawValue()**  
**current.get\_currentRawValue()**

**YCurrent**

---

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

---

**current→get\_currentValue()****YCurrent****current→currentValue()current.get\_currentValue()**

---

Returns the current measure for the current.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current measure for the current

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

**current→get\_errorMessage()** YCurrent  
**current→errorMessage()current.getErrorMessage()**

---

Returns the error message of the latest error with the current sensor.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the current sensor object

---

**current→get\_errorType()****YCurrent****current→errorType()current.get\_errorType()**

---

Returns the numerical error code of the latest error with the current sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the current sensor object

**current→get\_friendlyName()** **YCurrent**  
**current→friendlyName()current.get\_friendlyName()**

Returns a global identifier of the current sensor in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the current sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the current sensor (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the current sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**current→get\_functionDescriptor()**  
**current→functionDescriptor()**  
**current.get\_functionDescriptor()**

**YCurrent**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**current→get\_functionId()**

**YCurrent**

**current→functionId()current.get\_functionId()**

---

Returns the hardware identifier of the current sensor, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the current sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**current→get\_hardwareId()****YCurrent****current→hardwareId()current.get\_hardwareId()**

Returns the unique hardware identifier of the current sensor in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the current sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the current sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**current→get\_highestValue()**

**YCurrent**

**current→highestValue()current.get\_highestValue()**

---

Returns the maximal value observed for the current.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the current

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**current→get\_logFrequency()****YCurrent****current→logFrequency()current.get\_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**current→get\_logicalName()**

**YCurrent**

**current→logicalName()current.get\_logicalName()**

---

Returns the logical name of the current sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the current sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

---

**current→get\_lowestValue()****YCurrent****current→lowestValue()current.get\_lowestValue()**

---

Returns the minimal value observed for the current.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the current

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**current→get\_module()****YCurrent****current→module()current.get\_module()**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**current→get\_recordedData()****YCurrent****current→recordedData()current.get\_recordedData()**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**current→get\_reportFrequency()**  
**current→reportFrequency()**  
**current.get\_reportFrequency()**

**YCurrent**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**current→get\_resolution()****YCurrent****current→resolution()current.get\_resolution()**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**current→get\_unit()**

**YCurrent**

**current→unit()current.get\_unit()**

---

Returns the measuring unit for the current.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the current

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**current→get(userData)****YCurrent****current→userData()current.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**current→isOnline()current.isOnline()****YCurrent**

Checks if the current sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the current sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the current sensor.

**Returns :**

true if the current sensor can be reached, and false otherwise

**current→load()current.load()****YCurrent**

Preloads the current sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**current→loadCalibrationPoints()**  
**current.loadCalibrationPoints()****YCurrent**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**current→nextCurrent()current.nextCurrent()****YCurrent**

Continues the enumeration of current sensors started using `yFirstCurrent()`.

```
def nextCurrent( )
```

**Returns :**

a pointer to a `YCurrent` object, corresponding to a current sensor currently online, or a null pointer if there are no more current sensors to enumerate.

**current→registerTimedReportCallback()  
current.registerTimedReportCallback()****YCurrent**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**current→registerValueCallback()**  
**current.registerValueCallback()****YCurrent**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**current→set\_highestValue()**  
**current→setHighestValue()**  
**current.set\_highestValue()**

**YCurrent**

---

Changes the recorded maximal value observed pour the current.

```
def set_highestValue( newval )
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed pour the current

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**current→set\_logFrequency()**  
**current→setLogFrequency()**  
**current.set\_logFrequency()**

YCurrent

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**current→set\_logicalName()** YCurrent  
**current→setLogicalName()current.set\_logicalName()**

---

Changes the logical name of the current sensor.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the current sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**current→set\_lowestValue()****YCurrent****current→setLowestValue()current.set\_lowestValue()**

Changes the recorded minimal value observed pour the current.

```
def set_lowestValue( newval )
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed pour the current

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**current→set\_reportFrequency()**  
**current→setReportFrequency()**  
**current.set\_reportFrequency()**

**YCurrent**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**current→set\_resolution()****YCurrent****current→setResolution()current.set\_resolution()**

Changes the resolution of the measured values.

```
def set_resolution( newval )
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**current→set(userData)**

**YCurrent**

**current→setUserData()current.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.8. DataLogger function interface

Yoctopuce sensors include a non-volatile memory capable of storing ongoing measured data automatically, without requiring a permanent connection to a computer. The DataLogger function controls the global parameters of the internal data logger.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_datalogger.js'></script>
node.js	var yoctolib = require('yoctolib');
php	var YDataLogger = yoctolib.YDataLogger;
require_once('yocto_datalogger.php');	
cpp	#include "yocto_datalogger.h"
m	#import "yocto_datalogger.h"
pas	uses yocto_datalogger;
vb	yocto_datalogger.vb
cs	yocto_datalogger.cs
java	import com.yoctopuce.YoctoAPI.YDataLogger;
py	from yocto_datalogger import *

### Global functions

#### yFindDataLogger(func)

Retrieves a data logger for a given identifier.

#### yFirstDataLogger()

Starts the enumeration of data loggers currently accessible.

### YDataLogger methods

#### datalogger→describe()

Returns a short text that describes unambiguously the instance of the data logger in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

#### datalogger→forgetAllDataStreams()

Clears the data logger memory and discards all recorded data streams.

#### datalogger→get\_advertisedValue()

Returns the current value of the data logger (no more than 6 characters).

#### datalogger→get\_autoStart()

Returns the default activation state of the data logger on power up.

#### datalogger→get\_currentRunIndex()

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

#### datalogger→get\_dataSets()

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

#### datalogger→get\_dataStreams(v)

Builds a list of all data streams hold by the data logger (legacy method).

#### datalogger→get\_errorMessage()

Returns the error message of the latest error with the data logger.

#### datalogger→get\_errorType()

Returns the numerical error code of the latest error with the data logger.

#### datalogger→get\_friendlyName()

Returns a global identifier of the data logger in the format MODULE\_NAME . FUNCTION\_NAME.

#### datalogger→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

**datalogger→get\_functionId()**

Returns the hardware identifier of the data logger, without reference to the module.

**datalogger→get\_hardwareId()**

Returns the unique hardware identifier of the data logger in the form SERIAL . FUNCTIONID.

**datalogger→get\_logicalName()**

Returns the logical name of the data logger.

**datalogger→get\_module()**

Gets the YModule object for the device on which the function is located.

**datalogger→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**datalogger→get\_recording()**

Returns the current activation state of the data logger.

**datalogger→get\_timeUTC()**

Returns the Unix timestamp for current UTC time, if known.

**datalogger→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set(userData).

**datalogger→isOnline()**

Checks if the data logger is currently reachable, without raising any error.

**datalogger→isOnline\_async(callback, context)**

Checks if the data logger is currently reachable, without raising any error (asynchronous version).

**datalogger→load(msValidity)**

Preloads the data logger cache with a specified validity duration.

**datalogger→load\_async(msValidity, callback, context)**

Preloads the data logger cache with a specified validity duration (asynchronous version).

**datalogger→nextDataLogger()**

Continues the enumeration of data loggers started using yFirstDataLogger( ).

**datalogger→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**datalogger→set\_autoStart(newval)**

Changes the default activation state of the data logger on power up.

**datalogger→set\_logicalName(newval)**

Changes the logical name of the data logger.

**datalogger→set\_recording(newval)**

Changes the activation state of the data logger to start/stop recording data.

**datalogger→set\_timeUTC(newval)**

Changes the current UTC time reference used for recorded data.

**datalogger→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**datalogger→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**YDataLogger.FindDataLogger()****YDataLogger****yFindDataLogger()YDataLogger.FindDataLogger()**

Retrieves a data logger for a given identifier.

```
def FindDataLogger( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the data logger is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDataLogger.isOnline()` to test if the data logger is indeed online at a given time. In case of ambiguity when looking for a data logger by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the data logger

**Returns :**

a `YDataLogger` object allowing you to drive the data logger.

## **YDataLogger.FirstDataLogger()**

**YDataLogger**

## **yFirstDataLogger()YDataLogger.FirstDataLogger()**

---

Starts the enumeration of data loggers currently accessible.

```
def FirstDataLogger( )
```

Use the method `YDataLogger.nextDataLogger()` to iterate on next data loggers.

**Returns :**

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a null pointer if there are none.

**datalogger→describe()datalogger.describe()****YDataLogger**

Returns a short text that describes unambiguously the instance of the data logger in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the data logger (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**datalogger→forgetAllDataStreams()**  
**datalogger.forgetAllDataStreams()**

---

**YDataLogger**

Clears the data logger memory and discards all recorded data streams.

```
def forgetAllDataStreams( )
```

This method also resets the current run index to zero.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**datalogger→get\_advertisedValue()**  
**datalogger→advertisedValue()**  
**datalogger.get\_advertisedValue()**

**YDataLogger**

Returns the current value of the data logger (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the data logger (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**datalogger→get\_autoStart()**

**YDataLogger**

**datalogger→autoStart()datalogger.get\_autoStart()**

---

Returns the default activation state of the data logger on power up.

```
def get_autoStart( )
```

**Returns :**

either Y\_AUTOSTART\_OFF or Y\_AUTOSTART\_ON, according to the default activation state of the data logger on power up

On failure, throws an exception or returns Y\_AUTOSTART\_INVALID.

**datalogger→get\_currentRunIndex()**  
**datalogger→currentRunIndex()**  
**datalogger.get\_currentRunIndex()**

**YDataLogger**

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

```
def get_currentRunIndex( )
```

**Returns :**

an integer corresponding to the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point

On failure, throws an exception or returns Y\_CURRENTRUNINDEX\_INVALID.

**datalogger→get\_dataSets()****YDataLogger****datalogger→dataSets()datalogger.get\_dataSets()**

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

```
def get_dataSets( )
```

This function only works if the device uses a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

**Returns :**

a list of YDataSet object.

On failure, throws an exception or returns an empty list.

**datalogger→get\_dataStreams()**  
**datalogger→dataStreams()**  
**datalogger.get\_dataStreams()****YDataLogger**

Builds a list of all data streams hold by the data logger (legacy method).

```
def get_dataStreams( v)
```

The caller must pass by reference an empty array to hold YDataStream objects, and the function fills it with objects describing available data sequences.

This is the old way to retrieve data from the DataLogger. For new applications, you should rather use `get_dataSets()` method, or call directly `get_recordedData()` on the sensor object.

**Parameters :**

v an array of YDataStream objects to be filled in

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**datalogger→get\_errorMessage()**  
**datalogger→errorMessage()**  
**datalogger.get\_errorMessage()**

---

**YDataLogger**

Returns the error message of the latest error with the data logger.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the data logger object

---

**datalogger→get\_errorType()****YDataLogger****datalogger→errorType()datalogger.get\_errorType()**

---

Returns the numerical error code of the latest error with the data logger.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the data logger object

**datalogger→get\_friendlyName()**  
**datalogger→friendlyName()**  
**datalogger.get\_friendlyName()**

**YDataLogger**

Returns a global identifier of the data logger in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the data logger if they are defined, otherwise the serial number of the module and the hardware identifier of the data logger (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the data logger using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**datalogger→get\_functionDescriptor()**  
**datalogger→functionDescriptor()**  
**datalogger.get\_functionDescriptor()**

**YDataLogger**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**datalogger→get\_functionId()**

**YDataLogger**

**datalogger→functionId()datalogger.get\_functionId()**

---

Returns the hardware identifier of the data logger, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the data logger (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**datalogger→get\_hardwareId()**  
**datalogger→hardwareId()**  
**datalogger.get\_hardwareId()**

**YDataLogger**

Returns the unique hardware identifier of the data logger in the form SERIAL.FUNCTIONID.

**def get\_hardwareId( )**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the data logger. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the data logger (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**datalogger→get\_logicalName()**  
**datalogger→logicalName()**  
**datalogger.get\_logicalName()**

---

**YDataLogger**

Returns the logical name of the data logger.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the data logger. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

---

**datalogger→get\_module()****YDataLogger****datalogger→module()datalogger.get\_module()**

---

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

**datalogger→get\_recording()**

**YDataLogger**

**datalogger→recording()datalogger.get\_recording()**

---

Returns the current activation state of the data logger.

```
def get_recording( )
```

**Returns :**

either Y\_RECORDING\_OFF or Y\_RECORDING\_ON, according to the current activation state of the data logger

On failure, throws an exception or returns Y\_RECORDING\_INVALID.

**datalogger→get\_timeUTC()****YDataLogger****datalogger→timeUTC()datalogger.get\_timeUTC()**

Returns the Unix timestamp for current UTC time, if known.

```
def get_timeUTC( )
```

**Returns :**

an integer corresponding to the Unix timestamp for current UTC time, if known

On failure, throws an exception or returns Y\_TIMEUTC\_INVALID.

**datalogger→get(userData)**

**YDataLogger**

**datalogger→userData()datalogger.get(userData)**

---

Returns the value of the userData attribute, as previously stored using method set(userData).

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**datalogger→isOnline()datalogger.isOnline()****YDataLogger**

Checks if the data logger is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the data logger.

**Returns :**

`true` if the data logger can be reached, and `false` otherwise

**datalogger→load()datalogger.load()****YDataLogger**

Preloads the data logger cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**datalogger→nextDataLogger()**  
**datalogger.nextDataLogger()****YDataLogger**

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

```
def nextDataLogger( )
```

**Returns :**

a pointer to a `YDataLogger` object, corresponding to a data logger currently online, or a `null` pointer if there are no more data loggers to enumerate.

**datalogger→registerValueCallback()  
datalogger.registerValueCallback()****YDataLogger**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**datalogger→set\_autoStart()****YDataLogger****datalogger→setAutoStart()datalogger.set\_autoStart()**

Changes the default activation state of the data logger on power up.

```
def set_autoStart( newval)
```

Remember to call the saveToFlash( ) method of the module if the modification must be kept.

**Parameters :**

**newval** either Y\_AUTOSTART\_OFF or Y\_AUTOSTART\_ON, according to the default activation state of the data logger on power up

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**datalogger→set\_logicalName()**  
**datalogger→setLogicalName()**  
**datalogger.set\_logicalName()**

**YDataLogger**

Changes the logical name of the data logger.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the data logger.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**datalogger→set\_recording()**  
**datalogger→setRecording()**  
**datalogger.set\_recording()**

**YDataLogger**

Changes the activation state of the data logger to start/stop recording data.

```
def set_recording( newval)
```

**Parameters :**

**newval** either Y\_RECORDING\_OFF or Y\_RECORDING\_ON, according to the activation state of the data logger to start/stop recording data

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**datalogger→set\_timeUTC()**

**YDataLogger**

**datalogger→setTimeUTC()datalogger.set\_timeUTC()**

---

Changes the current UTC time reference used for recorded data.

```
def set_timeUTC( newval)
```

**Parameters :**

**newval** an integer corresponding to the current UTC time reference used for recorded data

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**datalogger→set(userData)****YDataLogger****datalogger→setUserData()datalogger.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData) data
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.9. Formatted data sequence

A run is a continuous interval of time during which a module was powered on. A data run provides easy access to all data collected during a given run, providing on-the-fly resampling at the desired reporting rate.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_datalogger.js'></script>
nodejs var yoctolib = require('yoctolib');
var YDataLogger = yoctolib.YDataLogger;
php require_once('yocto_datalogger.php');
cpp #include "yocto_datalogger.h"
m #import "yocto_datalogger.h"
pas uses yocto_datalogger;
vb yocto_datalogger.vb
cs yocto_datalogger.cs
java import com.yoctopuce.YoctoAPI.YDataLogger;
py from yocto_datalogger import *

```

### YDataRun methods

#### **datarun→get\_averageValue(measureName, pos)**

Returns the average value of the measure observed at the specified time period.

#### **datarun→get\_duration()**

Returns the duration (in seconds) of the data run.

#### **datarun→get\_maxValue(measureName, pos)**

Returns the maximal value of the measure observed at the specified time period.

#### **datarun→get\_measureNames()**

Returns the names of the measures recorded by the data logger.

#### **datarun→get\_minValue(measureName, pos)**

Returns the minimal value of the measure observed at the specified time period.

#### **datarun→get\_startTimeUTC()**

Returns the start time of the data run, relative to the Jan 1, 1970.

#### **datarun→get\_valueCount()**

Returns the number of values accessible in this run, given the selected data samples interval.

#### **datarun→get\_valueInterval()**

Returns the number of seconds covered by each value in this run.

#### **datarun→set\_valueInterval(valueInterval)**

Changes the number of seconds covered by each value in this run.

**datarun→get\_averageValue()****YDataRun****datarun→averageValue()datarun.get\_averageValue()**

Returns the average value of the measure observed at the specified time period.

```
def get_averageValue( measureName, pos)
```

**Parameters :**

**measureName** the name of the desired measure (one of the names returned by get\_measureNames)

**pos** the position index, between 0 and the value returned by get\_valueCount

**Returns :**

a floating point number (the average value)

On failure, throws an exception or returns Y\_AVERAGEVALUE\_INVALID.

**datarun→get\_duration()**

**YDataRun**

**datarun→duration()datarun.get\_duration()**

---

Returns the duration (in seconds) of the data run.

```
def get_duration( )
```

When the datalogger is actively recording and the specified run is the current run, calling this method reloads last sequence(s) from device to make sure it includes the latest recorded data.

**Returns :**

an unsigned number corresponding to the number of seconds between the beginning of the run (when the module was powered up) and the last recorded measure.

**datarun→get\_maxValue()****YDataRun****datarun→maxValue()datarun.get\_maxValue()**

Returns the maximal value of the measure observed at the specified time period.

```
def get_maxValue( measureName, pos)
```

**Parameters :**

**measureName** the name of the desired measure (one of the names returned by `get_measureNames`)

**pos** the position index, between 0 and the value returned by `get_valueCount`

**Returns :**

a floating point number (the maximal value)

On failure, throws an exception or returns Y\_MAXVALUE\_INVALID.

`datarun→get_measureNames()`  
`datarun→measureNames()`  
`datarun.get_measureNames()`

**YDataRun**

---

Returns the names of the measures recorded by the data logger.

`def get_measureNames( )`

In most case, the measure names match the hardware identifier of the sensor that produced the data.

**Returns :**

a list of strings (the measure names) On failure, throws an exception or returns an empty array.

**datarun→get\_minValue()****YDataRun****datarun→minValue()datarun.get\_minValue()**

Returns the minimal value of the measure observed at the specified time period.

```
def get_minValue( measureName, pos)
```

**Parameters :**

**measureName** the name of the desired measure (one of the names returned by get\_measureNames)

**pos** the position index, between 0 and the value returned by get\_valueCount

**Returns :**

a floating point number (the minimal value)

On failure, throws an exception or returns Y\_MINVALUE\_INVALID.

**datarun→get\_startTimeUTC()**  
**datarun→startTimeUTC()**

---

**YDataRun**

Returns the start time of the data run, relative to the Jan 1, 1970.

If the UTC time was not set in the datalogger at any time during the recording of this data run, and if this is not the current run, this method returns 0.

**Returns :**

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data run (i.e. Unix time representation of the absolute time).

**datarun→get\_valueCount()****YDataRun****datarun→valueCount()datarun.get\_valueCount()**

Returns the number of values accessible in this run, given the selected data samples interval.

```
def get_valueCount( )
```

When the datalogger is actively recording and the specified run is the current run, calling this method reloads last sequence(s) from device to make sure it includes the latest recorded data.

**Returns :**

an unsigned number corresponding to the run duration divided by the samples interval.

**datarun→get\_valueInterval()****YDataRun****datarun→valueInterval()datarun.get\_valueInterval()**

Returns the number of seconds covered by each value in this run.

```
def get_valueInterval( )
```

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.

**Returns :**

an unsigned number corresponding to a number of seconds covered by each data sample in the Run.

**datarun→set\_valueInterval()**  
**datarun→setValueInterval()**  
**datarun.set\_valueInterval()**

**YDataRun**

Changes the number of seconds covered by each value in this run.

```
def set_valueInterval( valueInterval)
```

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.

**Parameters :**

**valueInterval** an integer number of seconds.

**Returns :**

nothing

## 3.10. Recorded data sequence

YDataSet objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the YDataSet object is instanciated by the `get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the dataLogger.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_api.js'></script>
nodejs var yoctolib = require('yoctolib');
var YAPI = yoctolib.YAPI;
var YModule = yoctolib.YModule;
php require_once('yocto_api.php');
cpp #include "yocto_api.h"
m #import "yocto_api.h"
pas uses yocto_api;
vb yocto_api.vb
cs yocto_api.cs
java import com.yoctopuce.YoctoAPI.YModule;
py from yocto_api import *

```

### YDataSet methods

#### `dataset→get_endTimeUTC()`

Returns the end time of the dataset, relative to the Jan 1, 1970.

#### `dataset→get_functionId()`

Returns the hardware identifier of the function that performed the measure, without reference to the module.

#### `dataset→get_hardwareId()`

Returns the unique hardware identifier of the function who performed the measures, in the form SERIAL.FUNCTIONID.

#### `dataset→get_measures()`

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

#### `dataset→get_preview()`

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

#### `dataset→get_progress()`

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

#### `dataset→get_startTimeUTC()`

Returns the start time of the dataset, relative to the Jan 1, 1970.

#### `dataset→get_summary()`

Returns an YMeasure object which summarizes the whole DataSet.

#### `dataset→get_unit()`

Returns the measuring unit for the measured value.

**dataset→loadMore()**

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

**dataset→loadMore\_async(callback, context)**

Loads the the next block of measures from the dataLogger asynchronously.

**dataset→get\_endTimeUTC()****YDataSet****dataset→endTimeUTC()dataset.get\_endTimeUTC()**

Returns the end time of the dataset, relative to the Jan 1, 1970.

```
def get_endTimeUTC( )
```

When the YDataSet is created, the end time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

**Returns :**

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the end of this data set (i.e. Unix time representation of the absolute time).

---

**dataset→get\_functionId()****YDataSet****dataset→functionId()dataset.get\_functionId()**

---

Returns the hardware identifier of the function that performed the measure, without reference to the module.

```
def get_functionId( )
```

For example `temperature1`.

**Returns :**

a string that identifies the function (ex: `temperature1`)

**dataset→get\_hardwareId()**

**YDataSet**

**dataset→hardwareId()dataset.get\_hardwareId()**

---

Returns the unique hardware identifier of the function who performed the measures, in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example THRMCPL1-123456.temperature1)

**Returns :**

a string that uniquely identifies the function (ex: THRMCPL1-123456.temperature1)

On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**dataset→get\_measures()****YDataSet****dataset→measures()dataset.get\_measures()**

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

```
def get_measures( )
```

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `loadMore()` to load data from the device. You may have to call `loadMore()` several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

**Returns :**

a table of records, where each record depicts the measured value for a given time interval

On failure, throws an exception or returns an empty array.

**dataset→get\_preview()**

**YDataSet**

**dataset→preview()dataset.get\_preview()**

Returns a condensed version of the measures that can be retrieved in this YDataSet, as a list of YMeasure objects.

```
def get_preview( )
```

Each item includes: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This preview is available as soon as `loadMore( )` has been called for the first time.

**Returns :**

a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

---

**dataset→get\_progress()****YDataSet****dataset→progress()dataset.get\_progress()**

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

```
def get_progress( )
```

When the object is instanciated by `get_dataSet`, the progress is zero. Each time `loadMore()` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

**Returns :**

an integer in the range 0 to 100 (percentage of completion).

---

**dataset→getStartTimeUTC()** **YDataSet**  
**dataset→startTimeUTC()dataset.getStartTimeUTC()**

---

Returns the start time of the dataset, relative to the Jan 1, 1970.

```
def getStartTimeUTC( )
```

When the YDataSet is created, the start time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the start time is updated to reflect the timestamp of the first measure actually found in the dataLogger within the specified range.

**Returns :**

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data set (i.e. Unix time representation of the absolute time).

---

**dataset→get\_summary()****YDataSet****dataset→summary()dataset.get\_summary()**

---

Returns an YMeasure object which summarizes the whole DataSet.

```
def get_summary( )
```

In includes the following information: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This summary is available as soon as `loadMore( )` has been called for the first time.

**Returns :**

an YMeasure object

**dataset→get\_unit()**

**YDataSet**

**dataset→unit()dataset.get\_unit()**

---

Returns the measuring unit for the measured value.

```
def get_unit( )
```

**Returns :**

a string that represents a physical unit.

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**dataset→loadMore()dataset.loadMore()****YDataSet**

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

```
def loadMore( )
```

**Returns :**

an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

On failure, throws an exception or returns a negative error code.

## 3.11. Unformatted data sequence

YDataStream objects represent bare recorded measure sequences, exactly as found within the data logger present on Yoctopuce sensors.

In most cases, it is not necessary to use YDataStream objects directly, as the YDataSet objects (returned by the `get_recordedData()` method from sensors and the `get_dataSets()` method from the data logger) provide a more convenient interface.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_api.js'></script>
nodejs var yoctolib = require('yoctolib');
          var YAPI = yoctolib.YAPI;
          var YModule = yoctolib.YModule;
php require_once('yocto_api.php');
cpp #include "yocto_api.h"
m #import "yocto_api.h"
pas uses yocto_api;
vb yocto_api.vb
cs yocto_api.cs
java import com.yoctopuce.YoctoAPI.YModule;
py from yocto_api import *

```

### YDataStream methods

#### `datastream→get_averageValue()`

Returns the average of all measures observed within this stream.

#### `datastream→get_columnCount()`

Returns the number of data columns present in this stream.

#### `datastream→get_columnNames()`

Returns the title (or meaning) of each data column present in this stream.

#### `datastream→get_data(row, col)`

Returns a single measure from the data stream, specified by its row and column index.

#### `datastream→get_dataRows()`

Returns the whole data set contained in the stream, as a bidimensional table of numbers.

#### `datastream→get_dataSamplesIntervalMs()`

Returns the number of milliseconds between two consecutive rows of this data stream.

#### `datastream→get_duration()`

Returns the approximate duration of this stream, in seconds.

#### `datastream→get_maxValue()`

Returns the largest measure observed within this stream.

#### `datastream→get_minValue()`

Returns the smallest measure observed within this stream.

#### `datastream→getRowCount()`

Returns the number of data rows present in this stream.

#### `datastream→get_runIndex()`

Returns the run index of the data stream.

#### `datastream→get_startTime()`

Returns the relative start time of the data stream, measured in seconds.

#### `datastream→get_startTimeUTC()`

Returns the start time of the data stream, relative to the Jan 1, 1970.

**datastream→get\_averageValue()**  
**datastream→averageValue()**  
**datastream.get\_averageValue()**

**YDataStream**

Returns the average of all measures observed within this stream.

```
def get_averageValue( )
```

If the device uses a firmware older than version 13000, this method will always return Y\_DATA\_INVALID.

**Returns :**

a floating-point number corresponding to the average value, or Y\_DATA\_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y\_DATA\_INVALID.

**datastream→get\_columnCount()**  
**datastream→columnCount()**  
**datastream.get\_columnCount()**

**YDataStream**

Returns the number of data columns present in this stream.

```
def get_columnCount( )
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

**Returns :**

an unsigned number corresponding to the number of columns.

On failure, throws an exception or returns zero.

**datastream→get\_columnNames()**  
**datastream→columnNames()**  
**datastream.get\_columnNames()**

**YDataStream**

Returns the title (or meaning) of each data column present in this stream.

```
def get_columnNames( )
```

In most case, the title of the data column is the hardware identifier of the sensor that produced the data. For streams recorded at a lower recording rate, the dataLogger stores the min, average and max value during each measure interval into three columns with suffixes \_min, \_avg and \_max respectively.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

**Returns :**

a list containing as many strings as there are columns in the data stream.

On failure, throws an exception or returns an empty array.

**datastream→get\_data()****YDataStream****datastream→data()datastream.get\_data()**

Returns a single measure from the data stream, specified by its row and column index.

```
def get_data( row, col )
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

This method fetches the whole data stream from the device, if not yet done.

**Parameters :**

**row** row index

**col** column index

**Returns :**

a floating-point number

On failure, throws an exception or returns Y\_DATA\_INVALID.

**datastream→get\_dataRows()****YDataStream****datastream→dataRows()datastream.get\_dataRows()**

Returns the whole data set contained in the stream, as a bidimensional table of numbers.

```
def get_dataRows( )
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

This method fetches the whole data stream from the device, if not yet done.

**Returns :**

a list containing as many elements as there are rows in the data stream. Each row itself is a list of floating-point numbers.

On failure, throws an exception or returns an empty array.

**datastream→get\_dataSamplesIntervalMs()**  
**datastream→dataSamplesIntervalMs()**  
**datastream.get\_dataSamplesIntervalMs()**

**YDataStream**

Returns the number of milliseconds between two consecutive rows of this data stream.

```
def get_dataSamplesIntervalMs( )
```

By default, the data logger records one row per second, but the recording frequency can be changed for each device function

**Returns :**

an unsigned number corresponding to a number of milliseconds.

**datastream→get\_duration()**

**YDataStream**

**datastream→duration()datastream.get\_duration()**

---

Returns the approximate duration of this stream, in seconds.

```
def get_duration( )
```

**Returns :**

the number of seconds covered by this stream.

On failure, throws an exception or returns Y\_DURATION\_INVALID.

---

**datastream→get\_maxValue()****YDataStream****datastream→maxValue()datastream.get\_maxValue()**

---

Returns the largest measure observed within this stream.

```
def get_maxValue( )
```

If the device uses a firmware older than version 13000, this method will always return Y\_DATA\_INVALID.

**Returns :**

a floating-point number corresponding to the largest value, or Y\_DATA\_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y\_DATA\_INVALID.

---

**datastream→get\_minValue()** **YDataStream**  
**datastream→minValue()datastream.get\_minValue()**

---

Returns the smallest measure observed within this stream.

```
def get_minValue( )
```

If the device uses a firmware older than version 13000, this method will always return Y\_DATA\_INVALID.

**Returns :**

a floating-point number corresponding to the smallest value, or Y\_DATA\_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y\_DATA\_INVALID.

**datastream→getRowCount()****YDataStream****datastream→rowCount()datastream.getRowCount()**

Returns the number of data rows present in this stream.

```
def getRowCount( )
```

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

**Returns :**

an unsigned number corresponding to the number of rows.

On failure, throws an exception or returns zero.

**datastream→get\_runIndex()**

**YDataStream**

**datastream→runIndex()datastream.get\_runIndex()**

---

Returns the run index of the data stream.

```
def get_runIndex( )
```

A run can be made of multiple datastreams, for different time intervals.

**Returns :**

an unsigned number corresponding to the run index.

**datastream→getStartTime()****YDataStream****datastream→startTime()datastream.getStartTime()**

Returns the relative start time of the data stream, measured in seconds.

```
def getStartTime( )
```

For recent firmwares, the value is relative to the present time, which means the value is always negative. If the device uses a firmware older than version 13000, value is relative to the start of the time the device was powered on, and is always positive. If you need an absolute UTC timestamp, use `getStartTimeUTC()`.

**Returns :**

an unsigned number corresponding to the number of seconds between the start of the run and the beginning of this data stream.

**datastream→getStartTimeUTC()**  
**datastream→startTimeUTC()**  
**datastream.getStartTimeUTC()**

---

**YDataStream**

Returns the start time of the data stream, relative to the Jan 1, 1970.

```
def getStartTimeUTC( )
```

If the UTC time was not set in the datalogger at the time of the recording of this data stream, this method returns 0.

**Returns :**

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data stream (i.e. Unix time representation of the absolute time).

## 3.12. Digital IO function interface

The Yoctopuce application programming interface allows you to switch the state of each bit of the I/O port. You can switch all bits at once, or one by one. The library can also automatically generate short pulses of a determined duration. Electrical behavior of each I/O can be modified (open drain and reverse polarity).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_digitalio.js'></script>
node.js	var yoctolib = require('yoctolib');
	var YDigitalIO = yoctolib.YDigitalIO;
php	require_once('yocto_digitalio.php');
cpp	#include "yocto_digitalio.h"
m	#import "yocto_digitalio.h"
pas	uses yocto_digitalio;
vb	yocto_digitalio.vb
cs	yocto_digitalio.cs
java	import com.yoctopuce.YoctoAPI.YDigitalIO;
py	from yocto_digitalio import *

### Global functions

#### yFindDigitalIO(func)

Retrieves a digital IO port for a given identifier.

#### yFirstDigitalIO()

Starts the enumeration of digital IO ports currently accessible.

### YDigitalIO methods

#### digitalio→delayedPulse(bitno, ms\_delay, ms\_duration)

Schedules a pulse on a single bit for a specified duration.

#### digitalio→describe()

Returns a short text that describes unambiguously the instance of the digital IO port in the form TYPE (NAME) = SERIAL.FUNCTIONID.

#### digitalio→get\_advertisedValue()

Returns the current value of the digital IO port (no more than 6 characters).

#### digitalio→get\_bitDirection(bitno)

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

#### digitalio→get\_bitOpenDrain(bitno)

Returns the type of electrical interface of a single bit from the I/O port.

#### digitalio→get\_bitPolarity(bitno)

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

#### digitalio→get\_bitState(bitno)

Returns the state of a single bit of the I/O port.

#### digitalio→get\_errorMessage()

Returns the error message of the latest error with the digital IO port.

#### digitalio→get\_errorType()

Returns the numerical error code of the latest error with the digital IO port.

#### digitalio→get\_friendlyName()

Returns a global identifier of the digital IO port in the format MODULE\_NAME . FUNCTION\_NAME.

<b>digitalio→get_functionDescriptor()</b>	Returns a unique identifier of type YFUN_DESCR corresponding to the function.
<b>digitalio→get_functionId()</b>	Returns the hardware identifier of the digital IO port, without reference to the module.
<b>digitalio→get_hardwareId()</b>	Returns the unique hardware identifier of the digital IO port in the form SERIAL . FUNCTIONID.
<b>digitalio→get_logicalName()</b>	Returns the logical name of the digital IO port.
<b>digitalio→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>digitalio→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>digitalio→get_outputVoltage()</b>	Returns the voltage source used to drive output bits.
<b>digitalio→get_portDirection()</b>	Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.
<b>digitalio→get_portOpenDrain()</b>	Returns the electrical interface for each bit of the port.
<b>digitalio→get_portPolarity()</b>	Returns the polarity of all the bits of the port.
<b>digitalio→get_portSize()</b>	Returns the number of bits implemented in the I/O port.
<b>digitalio→get_portState()</b>	Returns the digital IO port state: bit 0 represents input 0, and so on.
<b>digitalio→get_userData()</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>digitalio→isOnline()</b>	Checks if the digital IO port is currently reachable, without raising any error.
<b>digitalio→isOnline_async(callback, context)</b>	Checks if the digital IO port is currently reachable, without raising any error (asynchronous version).
<b>digitalio→load(msValidity)</b>	Preloads the digital IO port cache with a specified validity duration.
<b>digitalio→load_async(msValidity, callback, context)</b>	Preloads the digital IO port cache with a specified validity duration (asynchronous version).
<b>digitalio→nextDigitalIO()</b>	Continues the enumeration of digital IO ports started using yFirstDigitalIO( ).
<b>digitalio→pulse(bitno, ms_duration)</b>	Triggers a pulse on a single bit for a specified duration.
<b>digitalio→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>digitalio→set_bitDirection(bitno, bitdirection)</b>	Changes the direction of a single bit from the I/O port.
<b>digitalio→set_bitOpenDrain(bitno, opendrain)</b>	Changes the electrical interface of a single bit from the I/O port.
<b>digitalio→set_bitPolarity(bitno, bitpolarity)</b>	

Changes the polarity of a single bit from the I/O port.

**digitalio→set\_bitState(bitno, bitstate)**

Sets a single bit of the I/O port.

**digitalio→set\_logicalName(newval)**

Changes the logical name of the digital IO port.

**digitalio→set\_outputVoltage(newval)**

Changes the voltage source used to drive output bits.

**digitalio→set\_portDirection(newval)**

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

**digitalio→set\_portOpenDrain(newval)**

Changes the electrical interface for each bit of the port.

**digitalio→set\_portPolarity(newval)**

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

**digitalio→set\_portState(newval)**

Changes the digital IO port state: bit 0 represents input 0, and so on.

**digitalio→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**digitalio→toggle\_bitState(bitno)**

Reverts a single bit of the I/O port.

**digitalio→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YDigitalIO.FindDigitalIO() yFindDigitalIO()YDigitalIO.FindDigitalIO()

YDigitalIO

Retrieves a digital IO port for a given identifier.

```
def FindDigitalIO( func )
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the digital IO port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDigitalIO.isOnline()` to test if the digital IO port is indeed online at a given time. In case of ambiguity when looking for a digital IO port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the digital IO port

### Returns :

a `YDigitalIO` object allowing you to drive the digital IO port.

**YDigitalIO.FirstDigitalIO()****YDigitalIO****yFirstDigitalIO()YDigitalIO.FirstDigitalIO()**

Starts the enumeration of digital IO ports currently accessible.

```
def FirstDigitalIO( )
```

Use the method `YDigitalIO.nextDigitalIO()` to iterate on next digital IO ports.

**Returns :**

a pointer to a `YDigitalIO` object, corresponding to the first digital IO port currently online, or a null pointer if there are none.

**digitalio→delayedPulse()**digitalio.delayedPulse()****

YDigitalIO

Schedules a pulse on a single bit for a specified duration.

```
def delayedPulse( bitno, ms_delay, ms_duration)
```

The specified bit will be turned to 1, and then back to 0 after the given duration.

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**ms\_delay** waiting time before the pulse, in milliseconds

**ms\_duration** desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**digitalio→describe()digitalio.describe()****YDigitalIO**

Returns a short text that describes unambiguously the instance of the digital IO port in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the digital IO port (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**digitalio→get\_advertisedValue()**  
**digitalio→advertisedValue()**  
**digitalio.get\_advertisedValue()**

**YDigitalIO**

---

Returns the current value of the digital IO port (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the digital IO port (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**digitalio→get\_bitDirection()****YDigitalIO****digitalio→bitDirection()digitalio.get\_bitDirection()**

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

```
def get_bitDirection( bitno )
```

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**digitalio→get\_bitOpenDrain()** YDigitalIO  
**digitalio→bitOpenDrain()digitalio.get\_bitOpenDrain()**

Returns the type of electrical interface of a single bit from the I/O port.

```
def get_bitOpenDrain( bitno)
```

(0 means the bit is an input, 1 an output).

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**Returns :**

0 means the a bit is a regular input/output, 1 means the bit is an open-drain (open-collector) input/output.

On failure, throws an exception or returns a negative error code.

**digitalio→get\_bitPolarity()****YDigitalIO****digitalio→bitPolarity()digitalio.get\_bitPolarity()**

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

```
def get_bitPolarity( bitno)
```

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**digitalio→get\_bitState()**

**YDigitalIO**

**digitalio→bitState()digitalio.get\_bitState()**

---

Returns the state of a single bit of the I/O port.

```
def get_bitState( bitno )
```

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**Returns :**

the bit state (0 or 1)

On failure, throws an exception or returns a negative error code.

**digitalio→get\_errorMessage()**  
**digitalio→errorMessage()**  
**digitalio.get\_errorMessage()**

**YDigitalIO**

Returns the error message of the latest error with the digital IO port.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the digital IO port object

**digitalio→get\_errorType()**

**YDigitalIO**

**digitalio→errorType()digitalio.get\_errorType()**

---

Returns the numerical error code of the latest error with the digital IO port.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the digital IO port object

**digitalio→get\_friendlyName()****YDigitalIO****digitalio→friendlyName()digitalio.get\_friendlyName()**

Returns a global identifier of the digital IO port in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the digital IO port if they are defined, otherwise the serial number of the module and the hardware identifier of the digital IO port (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the digital IO port using logical names (ex: MyCustomName . relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**digitalio→get\_functionDescriptor()**  
**digitalio→functionDescriptor()**  
**digitalio.get\_functionDescriptor()**

**YDigitalIO**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**digitalio→get\_functionId()****YDigitalIO****digitalio→functionId()digitalio.get\_functionId()**

Returns the hardware identifier of the digital IO port, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the digital IO port (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**digitalio→get\_hwrid()****YDigitalIO****digitalio→hwrid()digitalio.get\_hwrid()**

Returns the unique hardware identifier of the digital IO port in the form SERIAL.FUNCTIONID.

```
def get_hwrid( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the digital IO port. (for example RELAYLO1-123456.relay1)

**Returns :**

a string that uniquely identifies the digital IO port (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**digitalio→get\_logicalName()****YDigitalIO****digitalio→logicalName()digitalio.get\_logicalName()**

Returns the logical name of the digital IO port.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the digital IO port. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**digitalio→get\_module()**  
**digitalio→module()digitalio.get\_module()**

---

**YDigitalIO**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**digitalio→get\_outputVoltage()**  
**digitalio→outputVoltage()**  
**digitalio.get\_outputVoltage()**

**YDigitalIO**

Returns the voltage source used to drive output bits.

```
def get_outputVoltage( )
```

**Returns :**

a value among Y\_OUTPUTVOLTAGE\_USB\_5V, Y\_OUTPUTVOLTAGE\_USB\_3V and Y\_OUTPUTVOLTAGE\_EXT\_V corresponding to the voltage source used to drive output bits

On failure, throws an exception or returns Y\_OUTPUTVOLTAGE\_INVALID.

**digitalio→get\_portDirection()**

**YDigitalIO**

**digitalio→portDirection()digitalio.get\_portDirection()**

---

Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

```
def get_portDirection( )
```

**Returns :**

an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

On failure, throws an exception or returns Y\_PORTDIRECTION\_INVALID.

**digitalio→get\_portOpenDrain()**  
**digitalio→portOpenDrain()**  
**digitalio.get\_portOpenDrain()**

**YDigitalIO**

Returns the electrical interface for each bit of the port.

```
def get_portOpenDrain( )
```

For each bit set to 0 the matching I/O works in the regular, intuitive way, for each bit set to 1, the I/O works in reverse mode.

**Returns :**

an integer corresponding to the electrical interface for each bit of the port

On failure, throws an exception or returns Y\_PORTOPENDRAIN\_INVALID.

**digitalio→get\_portPolarity()**

**YDigitalIO**

**digitalio→portPolarity()digitalio.get\_portPolarity()**

---

Returns the polarity of all the bits of the port.

```
def get_portPolarity( )
```

For each bit set to 0, the matching I/O works the regular, intuitive way; for each bit set to 1, the I/O works in reverse mode.

**Returns :**

an integer corresponding to the polarity of all the bits of the port

On failure, throws an exception or returns Y\_PORTPOLARITY\_INVALID.

**digitalio→get\_portSize()****YDigitalIO****digitalio→portSize()digitalio.get\_portSize()**

Returns the number of bits implemented in the I/O port.

```
def get_portSize( )
```

**Returns :**

an integer corresponding to the number of bits implemented in the I/O port

On failure, throws an exception or returns Y\_PORTSIZE\_INVALID.

**digitalio→get\_portState()**

**YDigitalIO**

**digitalio→portState()digitalio.get\_portState()**

---

Returns the digital IO port state: bit 0 represents input 0, and so on.

```
def get_portState( )
```

**Returns :**

an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

On failure, throws an exception or returns Y\_PORTSTATE\_INVALID.

**digitalio→get(userData)****YDigitalIO****digitalio→userData()digitalio.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**digitalio→isOnline()digitalio.isOnline()****YDigitalIO**

Checks if the digital IO port is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the digital IO port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the digital IO port.

**Returns :**

`true` if the digital IO port can be reached, and `false` otherwise

**digitalio→load()  
digitalio.load()****YDigitalIO**

Preloads the digital IO port cache with a specified validity duration.

```
def load( msValidity )
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## **digitalio→nextDigitalIO()digitalio.nextDigitalIO()**

**YDigitalIO**

Continues the enumeration of digital IO ports started using `yFirstDigitalIO()`.

```
def nextDigitalIO( )
```

**Returns :**

a pointer to a `YDigitalIO` object, corresponding to a digital IO port currently online, or a `null` pointer if there are no more digital IO ports to enumerate.

**digitalio→pulse()**digitalio.pulse()******YDigitalIO**

Triggers a pulse on a single bit for a specified duration.

```
def pulse( bitno, ms_duration)
```

The specified bit will be turned to 1, and then back to 0 after the given duration.

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**ms\_duration** desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**digitalio→registerValueCallback()  
digitalio.registerValueCallback()****YDigitalIO**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**digitalio→set\_bitDirection()****YDigitalIO****digitalio→setBitDirection()digitalio.set\_bitDirection()**

Changes the direction of a single bit from the I/O port.

```
def set_bitDirection( bitno, bitdirection)
```

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**bitdirection** direction to set, 0 makes the bit an input, 1 makes it an output. Remember to call the `saveToFlash( )` method to make sure the setting is kept after a reboot.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**digitalio→set\_bitOpenDrain()**  
**digitalio→setBitOpenDrain()**  
**digitalio.set\_bitOpenDrain()**

**YDigitalIO**

Changes the electrical interface of a single bit from the I/O port.

```
def set_bitOpenDrain( bitno, opendrain)
```

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**opendrain** 0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output.  
Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**digitalio→set\_bitPolarity()****YDigitalIO****digitalio→setBitPolarity()digitalio.set\_bitPolarity()**

Changes the polarity of a single bit from the I/O port.

```
def set_bitPolarity( bitno, bitpolarity)
```

**Parameters :**

**bitno** the bit number; lowest bit has index 0.

**bitpolarity** polarity to set, 0 makes the I/O work in regular mode, 1 makes the I/O works in reverse mode.

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**digitalio→set\_bitState()**  
**digitalio→setBitState()digitalio.set\_bitState()****YDigitalIO**

Sets a single bit of the I/O port.

```
def set_bitState( bitno, bitstate)
```

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**bitstate** the state of the bit (1 or 0)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**digitalio→set\_logicalName()**  
**digitalio→setLogicalName()**  
**digitalio.set\_logicalName()**

**YDigitalIO**

Changes the logical name of the digital IO port.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the digital IO port.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**digitalio→set\_outputVoltage()**  
**digitalio→setOutputVoltage()**  
**digitalio.set\_outputVoltage()**

**YDigitalIO**

Changes the voltage source used to drive output bits.

```
def set_outputVoltage( newval)
```

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

**Parameters :**

**newval** a value among `Y_OUTPUTVOLTAGE_USB_5V`, `Y_OUTPUTVOLTAGE_USB_3V` and `Y_OUTPUTVOLTAGE_EXT_V` corresponding to the voltage source used to drive output bits

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**digitalio→set\_portDirection()**  
**digitalio→setPortDirection()**  
**digitalio.set\_portDirection()****YDigitalIO**

Changes the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

```
def set_portDirection( newval)
```

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

**Parameters :**

**newval** an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

`digitalio→set_portOpenDrain()`  
`digitalio→setPortOpenDrain()`  
`digitalio.set_portOpenDrain()`

YDigitalIO

Changes the electrical interface for each bit of the port.

```
def set_portOpenDrain( newval)
```

0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash( )` method to make sure the setting is kept after a reboot.

**Parameters :**

`newval` an integer corresponding to the electrical interface for each bit of the port

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**digitalio→set\_portPolarity()****YDigitalIO****digitalio→setPortPolarity()digitalio.set\_portPolarity()**

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

```
def set_portPolarity( newval)
```

Remember to call the `saveToFlash()` method to make sure the setting will be kept after a reboot.

**Parameters :**

**newval** an integer corresponding to the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**digitalio→set\_portState()** YDigitalIO  
**digitalio→setPortState()digitalio.set\_portState()**

Changes the digital IO port state: bit 0 represents input 0, and so on.

```
def set_portState( newval)
```

This function has no effect on bits configured as input in `portDirection`.

**Parameters :**

`newval` an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**digitalio→set(userData)****YDigitalIO****digitalio→setUserData()digitalio.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

**digitalio→toggle\_bitState()|digitalio.toggle\_bitState()****YDigitalIO**

Reverts a single bit of the I/O port.

```
def toggle_bitState( bitno)
```

**Parameters :**

**bitno** the bit number; lowest bit has index 0

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## 3.13. Display function interface

Yoctopuce display interface has been designed to easily show information and images. The device provides built-in multi-layer rendering. Layers can be drawn offline, individually, and freely moved on the display. It can also replay recorded sequences (animations).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_display.js'></script>
node.js	var yoctolib = require('yoctolib');
	var YDisplay = yoctolib.YDisplay;
php	require_once('yocto_display.php');
cpp	#include "yocto_display.h"
m	#import "yocto_display.h"
pas	uses yocto_display;
vb	yocto_display.vb
cs	yocto_display.cs
java	import com.yoctopuce.YoctoAPI.YDisplay;
py	from yocto_display import *

### Global functions

#### yFindDisplay(func)

Retrieves a display for a given identifier.

#### yFirstDisplay()

Starts the enumeration of displays currently accessible.

### YDisplay methods

#### display→copyLayerContent(srcLayerId, dstLayerId)

Copies the whole content of a layer to another layer.

#### display→describe()

Returns a short text that describes unambiguously the instance of the display in the form TYPE (NAME )=SERIAL.FUNCTIONID.

#### display→fade(brightness, duration)

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

#### display→get\_advertisedValue()

Returns the current value of the display (no more than 6 characters).

#### display→get\_brightness()

Returns the luminosity of the module informative leds (from 0 to 100).

#### display→get\_displayHeight()

Returns the display height, in pixels.

#### display→get\_displayLayer(layerId)

Returns a YDisplayLayer object that can be used to draw on the specified layer.

#### display→get\_displayType()

Returns the display type: monochrome, gray levels or full color.

#### display→get\_displayWidth()

Returns the display width, in pixels.

#### display→get\_enabled()

Returns true if the screen is powered, false otherwise.

#### display→get\_errorMessage()

Returns the error message of the latest error with the display.

### 3. Reference

<b>display-&gt;get_errorType()</b>
Returns the numerical error code of the latest error with the display.
<b>display-&gt;get_friendlyName()</b>
Returns a global identifier of the display in the format MODULE_NAME . FUNCTION_NAME.
<b>display-&gt;get_functionDescriptor()</b>
Returns a unique identifier of type YFUN_DESCR corresponding to the function.
<b>display-&gt;get_functionId()</b>
Returns the hardware identifier of the display, without reference to the module.
<b>display-&gt;get_hardwareId()</b>
Returns the unique hardware identifier of the display in the form SERIAL . FUNCTIONID.
<b>display-&gt;get_layerCount()</b>
Returns the number of available layers to draw on.
<b>display-&gt;get_layerHeight()</b>
Returns the height of the layers to draw on, in pixels.
<b>display-&gt;get_layerWidth()</b>
Returns the width of the layers to draw on, in pixels.
<b>display-&gt;get_logicalName()</b>
Returns the logical name of the display.
<b>display-&gt;get_module()</b>
Gets the YModule object for the device on which the function is located.
<b>display-&gt;get_module_async(callback, context)</b>
Gets the YModule object for the device on which the function is located (asynchronous version).
<b>display-&gt;get_orientation()</b>
Returns the currently selected display orientation.
<b>display-&gt;get_startupSeq()</b>
Returns the name of the sequence to play when the displayed is powered on.
<b>display-&gt;get(userData)</b>
Returns the value of the userData attribute, as previously stored using method set(userData).
<b>display-&gt;isOnline()</b>
Checks if the display is currently reachable, without raising any error.
<b>display-&gt;isOnline_async(callback, context)</b>
Checks if the display is currently reachable, without raising any error (asynchronous version).
<b>display-&gt;load(msValidity)</b>
Preloads the display cache with a specified validity duration.
<b>display-&gt;load_async(msValidity, callback, context)</b>
Preloads the display cache with a specified validity duration (asynchronous version).
<b>display-&gt;newSequence()</b>
Starts to record all display commands into a sequence, for later replay.
<b>display-&gt;nextDisplay()</b>
Continues the enumeration of displays started using yFirstDisplay( ).
<b>display-&gt;pauseSequence(delay_ms)</b>
Waits for a specified delay (in milliseconds) before playing next commands in current sequence.
<b>display-&gt;playSequence(sequenceName)</b>
Replays a display sequence previously recorded using newSequence( ) and saveSequence( ).
<b>display-&gt;registerValueCallback(callback)</b>

Registers the callback function that is invoked on every change of advertised value.

**display→resetAll()**

Clears the display screen and resets all display layers to their default state.

**display→saveSequence(sequenceName)**

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

**display→set\_brightness(newval)**

Changes the brightness of the display.

**display→set\_enabled(newval)**

Changes the power state of the display.

**display→set\_logicalName(newval)**

Changes the logical name of the display.

**display→set\_orientation(newval)**

Changes the display orientation.

**display→set\_startupSeq(newval)**

Changes the name of the sequence to play when the displayed is powered on.

**display→set(userData)**

Stores a user context provided as argument in the userData attribute of the function.

**display→stopSequence()**

Stops immediately any ongoing sequence replay.

**display→swapLayerContent(layerIdA, layerIdB)**

Swaps the whole content of two layers.

**display→upload(pathname, content)**

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

**display→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YDisplay.FindDisplay() yFindDisplay()YDisplay.FindDisplay()

YDisplay

Retrieves a display for a given identifier.

```
def FindDisplay( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the display is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDisplay.isOnline()` to test if the display is indeed online at a given time. In case of ambiguity when looking for a display by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the display

### Returns :

a `YDisplay` object allowing you to drive the display.

**YDisplay.FirstDisplay()****YDisplay****yFirstDisplay()YDisplay.FirstDisplay()**

Starts the enumeration of displays currently accessible.

```
def FirstDisplay( )
```

Use the method `YDisplay.nextDisplay()` to iterate on next displays.

**Returns :**

a pointer to a `YDisplay` object, corresponding to the first display currently online, or a `null` pointer if there are none.

**display→copyLayerContent()  
display.copyLayerContent()****YDisplay**

Copies the whole content of a layer to another layer.

```
def copyLayerContent( srcLayerId, dstLayerId)
```

The color and transparency of all the pixels from the destination layer are set to match the source pixels. This method only affects the displayed content, but does not change any property of the layer object. Note that layer 0 has no transparency support (it is always completely opaque).

**Parameters :**

**srcLayerId** the identifier of the source layer (a number in range 0..layerCount-1)

**dstLayerId** the identifier of the destination layer (a number in range 0..layerCount-1)

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**display→describe()display.describe()****YDisplay**

Returns a short text that describes unambiguously the instance of the display in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the display (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**display→fade()display.fade()****YDisplay**

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

```
def fade( brightness, duration)
```

**Parameters :**

**brightness** the new screen brightness

**duration** duration of the brightness transition, in milliseconds.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**display→get\_advertisedValue()**  
**display→advertisedValue()**  
**display.get\_advertisedValue()**

**YDisplay**

Returns the current value of the display (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the display (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**display→get\_brightness()**

**YDisplay**

**display→brightness()display.get\_brightness()**

---

Returns the luminosity of the module informative leds (from 0 to 100).

```
def get_brightness( )
```

**Returns :**

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns Y\_BRIGHTNESS\_INVALID.

**display→get\_displayHeight()****YDisplay****display→displayHeight()display.get\_displayHeight()**

Returns the display height, in pixels.

```
def get_displayHeight( )
```

**Returns :**

an integer corresponding to the display height, in pixels

On failure, throws an exception or returns Y\_DISPLAYHEIGHT\_INVALID.

---

**display→get\_displayLayer()** **YDisplay**  
**display→displayLayer()display.get\_displayLayer()**

Returns a YDisplayLayer object that can be used to draw on the specified layer.

```
def get_displayLayer( layerId )
```

The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

**Parameters :**

**layerId** the identifier of the layer (a number in range 0..layerCount-1)

**Returns :**

an YDisplayLayer object

On failure, throws an exception or returns null.

**display→get\_displayType()****YDisplay****display→displayType()display.get\_displayType()**

Returns the display type: monochrome, gray levels or full color.

```
def get_displayType( )
```

**Returns :**

a value among Y\_DISPLAYTYPE\_MONO, Y\_DISPLAYTYPE\_GRAY and Y\_DISPLAYTYPE\_RGB corresponding to the display type: monochrome, gray levels or full color

On failure, throws an exception or returns Y\_DISPLAYTYPE\_INVALID.

**display→get\_displayWidth()**

**YDisplay**

**display→displayWidth()display.get\_displayWidth()**

---

Returns the display width, in pixels.

```
def get_displayWidth( )
```

**Returns :**

an integer corresponding to the display width, in pixels

On failure, throws an exception or returns Y\_DISPLAYWIDTH\_INVALID.

**display→get\_enabled()****YDisplay****display→enabled()display.get\_enabled()**

Returns true if the screen is powered, false otherwise.

```
def get_enabled( )
```

**Returns :**

either Y\_ENABLED\_FALSE or Y\_ENABLED\_TRUE, according to true if the screen is powered, false otherwise

On failure, throws an exception or returns Y\_ENABLED\_INVALID.

**display→get\_errorMessage()** YDisplay  
**display→errorMessage()display.getErrorMessage()**

---

Returns the error message of the latest error with the display.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the display object

**display→get\_errorType()****YDisplay****display→errorType()display.get\_errorType()**

Returns the numerical error code of the latest error with the display.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the display object

**display→get\_friendlyName()** YDisplay

**display→friendlyName()display.get\_friendlyName()**

---

Returns a global identifier of the display in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the display if they are defined, otherwise the serial number of the module and the hardware identifier of the display (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the display using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**display→get\_functionDescriptor()**  
**display→functionDescriptor()**  
**display.get\_functionDescriptor()**

**YDisplay**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**display→get\_functionId()**  
**display→functionId()display.get\_functionId()**

---

**YDisplay**

Returns the hardware identifier of the display, without reference to the module.

```
def get_functionId( )
```

For example relay1

**Returns :**

a string that identifies the display (ex: relay1) On failure, throws an exception or returns Y\_FUNCTIONID\_INVALID.

---

**display→get\_hardwareId()****YDisplay****display→hardwareId()display.get\_hardwareId()**

---

Returns the unique hardware identifier of the display in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the display. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the display (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**display→get\_layerCount()**

**YDisplay**

**display→layerCount()display.get\_layerCount()**

---

Returns the number of available layers to draw on.

```
def get_layerCount( )
```

**Returns :**

an integer corresponding to the number of available layers to draw on

On failure, throws an exception or returns Y\_LAYERCOUNT\_INVALID.

**display→get\_layerHeight()  
display→layerHeight()display.get\_layerHeight()****YDisplay**

Returns the height of the layers to draw on, in pixels.

```
def get_layerHeight( )
```

**Returns :**

an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y\_LAYERHEIGHT\_INVALID.

**display→get\_layerWidth()**

**YDisplay**

**display→layerWidth()display.get\_layerWidth()**

---

Returns the width of the layers to draw on, in pixels.

```
def get_layerWidth( )
```

**Returns :**

an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns Y\_LAYERWIDTH\_INVALID.

**display→get\_logicalName()****YDisplay****display→logicalName()display.get\_logicalName()**

Returns the logical name of the display.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the display. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

---

<b>display→get_module()</b>	<b>YDisplay</b>
<b>display→module()display.get_module()</b>	

---

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**display→get\_orientation()****YDisplay****display→orientation()display.get\_orientation()**

Returns the currently selected display orientation.

```
def get_orientation( )
```

**Returns :**

a value among Y\_ORIENTATION\_LEFT, Y\_ORIENTATION\_UP, Y\_ORIENTATION\_RIGHT and Y\_ORIENTATION\_DOWN corresponding to the currently selected display orientation

On failure, throws an exception or returns Y\_ORIENTATION\_INVALID.

**display→get\_startupSeq()**

**YDisplay**

**display→startupSeq()display.get\_startupSeq()**

---

Returns the name of the sequence to play when the displayed is powered on.

```
def get_startupSeq( )
```

**Returns :**

a string corresponding to the name of the sequence to play when the displayed is powered on

On failure, throws an exception or returns Y\_STARTUPSEQ\_INVALID.

---

**display→get(userData)****YDisplay****display→userData()display.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**display→isOnline()display.isOnline()****YDisplay**

Checks if the display is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the display in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the display.

**Returns :**

`true` if the display can be reached, and `false` otherwise

**display→load()display.load()****YDisplay**

Preloads the display cache with a specified validity duration.

```
def load( msValidity )
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**display→newSequence()**display.newSequence()******YDisplay**

Starts to record all display commands into a sequence, for later replay.

```
def newSequence( )
```

The name used to store the sequence is specified when calling `saveSequence()`, once the recording is complete.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**display→nextDisplay()display.nextDisplay()****YDisplay**

Continues the enumeration of displays started using `yFirstDisplay()`.

```
def nextDisplay( )
```

**Returns :**

a pointer to a `YDisplay` object, corresponding to a display currently online, or a `null` pointer if there are no more displays to enumerate.

**display→pauseSequence()  
display.pauseSequence()****YDisplay**

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

```
def pauseSequence( delay_ms )
```

This method can be used while recording a display sequence, to insert a timed wait in the sequence (without any immediate effect). It can also be used dynamically while playing a pre-recorded sequence, to suspend or resume the execution of the sequence. To cancel a delay, call the same method with a zero delay.

**Parameters :**

**delay\_ms** the duration to wait, in milliseconds

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**display→playSequence()  
display.playSequence()****YDisplay**

Replays a display sequence previously recorded using newSequence( ) and saveSequence( ).

```
def playSequence( sequenceName)
```

**Parameters :**

**sequenceName** the name of the newly created sequence

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**display→registerValueCallback()  
display.registerValueCallback()****YDisplay**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**display→resetAll()display.resetAll()****YDisplay**

Clears the display screen and resets all display layers to their default state.

```
def resetAll( )
```

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**display→saveSequence()display.saveSequence()****YDisplay**

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

```
def saveSequence( sequenceName)
```

The sequence can be later replayed using `playSequence()`.

**Parameters :**

**sequenceName** the name of the newly created sequence

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**display→set\_brightness()  
display→setBrightness()display.set\_brightness()****YDisplay**

Changes the brightness of the display.

```
def set_brightness( newval)
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** an integer corresponding to the brightness of the display

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**display→set\_enabled()** YDisplay  
**display→setEnabled()display.set\_enabled()**

---

Changes the power state of the display.

```
def set_enabled( newval )
```

**Parameters :**

**newval** either Y\_ENABLED\_FALSE or Y\_ENABLED\_TRUE, according to the power state of the display

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**display→set\_logicalName()** **YDisplay**  
**display→setLogicalName()display.set\_logicalName()**

---

Changes the logical name of the display.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the display.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

---

**display→set\_orientation()  
display→setOrientation()display.set\_orientation()****YDisplay**

Changes the display orientation.

```
def set_orientation( newval)
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a value among `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_ORIENTATION_DOWN` corresponding to the display orientation

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**display→set\_startupSeq()****YDisplay****display→setStartupSeq()display.set\_startupSeq()**

Changes the name of the sequence to play when the displayed is powered on.

```
def set_startupSeq( newval)
```

Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the name of the sequence to play when the displayed is powered on

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**display→set(userData)**

**YDisplay**

**display→setUserData()display.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

**display→stopSequence()display.stopSequence()****YDisplay**

Stops immediately any ongoing sequence replay.

```
def stopSequence( )
```

The display is left as is.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**display→swapLayerContent()  
display.swapLayerContent()****YDisplay**

Swaps the whole content of two layers.

```
def swapLayerContent( layerIdA, layerIdB )
```

The color and transparency of all the pixels from the two layers are swapped. This method only affects the displayed content, but does not change any property of the layer objects. In particular, the visibility of each layer stays unchanged. When used between one hidden layer and a visible layer, this method makes it possible to easily implement double-buffering. Note that layer 0 has no transparency support (it is always completely opaque).

**Parameters :**

**layerIdA** the first layer (a number in range 0..layerCount-1)

**layerIdB** the second layer (a number in range 0..layerCount-1)

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**display→upload()**display.upload()******YDisplay**

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

```
def upload( pathname, content)
```

If a file already exists with the same path name, its content is overwritten.

**Parameters :**

**pathname** path and name of the new file to create

**content** binary buffer with the content to set

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## 3.14. DisplayLayer object interface

A DisplayLayer is an image layer containing objects to display (bitmaps, text, etc.). The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_display.js'></script>
nodejs var yoctolib = require('yoctolib');
var YDisplay = yoctolib.YDisplay;
require_once('yocto_display.php');
#include "yocto_display.h"
m #import "yocto_display.h"
pas uses yocto_display;
vb yocto_display.vb
cs yocto_display.cs
java import com.yoctopuce.YoctoAPI.YDisplay;
py from yocto_display import *

```

### YDisplayLayer methods

#### **displaylayer→clear()**

Erases the whole content of the layer (makes it fully transparent).

#### **displaylayer→clearConsole()**

Banks the console area within console margins, and resets the console pointer to the upper left corner of the console.

#### **displaylayer→consoleOut(text)**

Outputs a message in the console area, and advances the console pointer accordingly.

#### **displaylayer→drawBar(x1, y1, x2, y2)**

Draws a filled rectangular bar at a specified position.

#### **displaylayer→drawBitmap(x, y, w, bitmap, bgcol)**

Draws a bitmap at the specified position.

#### **displaylayer→drawCircle(x, y, r)**

Draws an empty circle at a specified position.

#### **displaylayer→drawDisc(x, y, r)**

Draws a filled disc at a given position.

#### **displaylayer→drawImage(x, y, imagename)**

Draws a GIF image at the specified position.

#### **displaylayer→drawPixel(x, y)**

Draws a single pixel at the specified position.

#### **displaylayer→drawRect(x1, y1, x2, y2)**

Draws an empty rectangle at a specified position.

#### **displaylayer→drawText(x, y, anchor, text)**

Draws a text string at the specified position.

#### **displaylayer→get\_display()**

Gets parent YDisplay.

#### **displaylayer→get\_displayHeight()**

Returns the display height, in pixels.

#### **displaylayer→get\_displayWidth()**

Returns the display width, in pixels.

**displaylayer→get\_layerHeight()**

Returns the height of the layers to draw on, in pixels.

**displaylayer→get\_layerWidth()**

Returns the width of the layers to draw on, in pixels.

**displaylayer→hide()**

Hides the layer.

**displaylayer→lineTo(x, y)**

Draws a line from current drawing pointer position to the specified position.

**displaylayer→moveTo(x, y)**

Moves the drawing pointer of this layer to the specified position.

**displaylayer→reset()**

Reverts the layer to its initial state (fully transparent, default settings).

**displaylayer→selectColorPen(color)**

Selects the pen color for all subsequent drawing functions, including text drawing.

**displaylayer→selectEraser()**

Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

**displaylayer→selectFont(fontname)**

Selects a font to use for the next text drawing functions, by providing the name of the font file.

**displaylayer→selectGrayPen(graylevel)**

Selects the pen gray level for all subsequent drawing functions, including text drawing.

**displaylayer→setAntialiasingMode(mode)**

Enables or disables anti-aliasing for drawing oblique lines and circles.

**displaylayer→setConsoleBackground(bgcol)**

Sets up the background color used by the clearConsole function and by the console scrolling feature.

**displaylayer→setConsoleMargins(x1, y1, x2, y2)**

Sets up display margins for the consoleOut function.

**displaylayer→setConsoleWordWrap(wordwrap)**

Sets up the wrapping behaviour used by the consoleOut function.

**displaylayer→setLayerPosition(x, y, scrollTime)**

Sets the position of the layer relative to the display upper left corner.

**displaylayer→unhide()**

Shows the layer.

**displaylayer→clear()  
displaylayer.clear()****YDisplayLayer**

Erases the whole content of the layer (makes it fully transparent).

```
def clear( )
```

This method does not change any other attribute of the layer. To reinitialize the layer attributes to defaults settings, use the method `reset( )` instead.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**displaylayer→clearConsole()**  
**displaylayer.clearConsole()****YDisplayLayer**

Banks the console area within console margins, and resets the console pointer to the upper left corner of the console.

```
def clearConsole( )
```

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→consoleOut()displaylayer.consoleOut()****YDisplayLayer**

Outputs a message in the console area, and advances the console pointer accordingly.

```
def consoleOut( text)
```

The console pointer position is automatically moved to the beginning of the next line when a newline character is met, or when the right margin is hit. When the new text to display extends below the lower margin, the console area is automatically scrolled up.

**Parameters :**

**text** the message to display

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→drawBar()displaylayer.drawBar()****YDisplayLayer**

Draws a filled rectangular bar at a specified position.

```
def drawBar( x1, y1, x2, y2)
```

**Parameters :**

**x1** the distance from left of layer to the left border of the rectangle, in pixels

**y1** the distance from top of layer to the top border of the rectangle, in pixels

**x2** the distance from left of layer to the right border of the rectangle, in pixels

**y2** the distance from top of layer to the bottom border of the rectangle, in pixels

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→drawBitmap()**  
**displaylayer.drawBitmap()****YDisplayLayer**

Draws a bitmap at the specified position.

```
def drawBitmap( x, y, w, bitmap, bgcol)
```

The bitmap is provided as a binary object, where each pixel maps to a bit, from left to right and from top to bottom. The most significant bit of each byte maps to the leftmost pixel, and the least significant bit maps to the rightmost pixel. Bits set to 1 are drawn using the layer selected pen color. Bits set to 0 are drawn using the specified background gray level, unless -1 is specified, in which case they are not drawn at all (as if transparent).

**Parameters :**

- x** the distance from left of layer to the left of the bitmap, in pixels
- y** the distance from top of layer to the top of the bitmap, in pixels
- w** the width of the bitmap, in pixels
- bitmap** a binary object
- bgcol** the background gray level to use for zero bits (0 = black, 255 = white), or -1 to leave the pixels unchanged

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→drawCircle()displaylayer.drawCircle()****YDisplayLayer**

Draws an empty circle at a specified position.

```
def drawCircle( x, y, r)
```

**Parameters :**

**x** the distance from left of layer to the center of the circle, in pixels

**y** the distance from top of layer to the center of the circle, in pixels

**r** the radius of the circle, in pixels

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→drawDisc()  
displaylayer.drawDisc()****YDisplayLayer**

Draws a filled disc at a given position.

```
def drawDisc( x, y, r)
```

**Parameters :**

**x** the distance from left of layer to the center of the disc, in pixels

**y** the distance from top of layer to the center of the disc, in pixels

**r** the radius of the disc, in pixels

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→drawImage()displaylayer.drawImage()****YDisplayLayer**

Draws a GIF image at the specified position.

```
def drawImage( x, y, imagename)
```

The GIF image must have been previously uploaded to the device built-in memory. If you experience problems using an image file, check the device logs for any error message such as missing image file or bad image file format.

**Parameters :**

- x** the distance from left of layer to the left of the image, in pixels
- y** the distance from top of layer to the top of the image, in pixels
- imagename** the GIF file name

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→drawPixel()displaylayer.drawPixel()****YDisplayLayer**

Draws a single pixel at the specified position.

```
def drawPixel( x, y )
```

**Parameters :**

**x** the distance from left of layer, in pixels

**y** the distance from top of layer, in pixels

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→drawRect()displaylayer.drawRect()****YDisplayLayer**

Draws an empty rectangle at a specified position.

```
def drawRect( x1, y1, x2, y2)
```

**Parameters :**

**x1** the distance from left of layer to the left border of the rectangle, in pixels

**y1** the distance from top of layer to the top border of the rectangle, in pixels

**x2** the distance from left of layer to the right border of the rectangle, in pixels

**y2** the distance from top of layer to the bottom border of the rectangle, in pixels

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→drawText()displaylayer.drawText()****YDisplayLayer**

Draws a text string at the specified position.

```
def drawText( x, y, anchor, text)
```

The point of the text that is aligned to the specified pixel position is called the anchor point, and can be chosen among several options. Text is rendered from left to right, without implicit wrapping.

**Parameters :**

**x** the distance from left of layer to the text anchor point, in pixels  
**y** the distance from top of layer to the text anchor point, in pixels  
**anchor** the text anchor point, chosen among the Y\_ALIGN enumeration: Y\_ALIGN\_TOP\_LEFT, Y\_ALIGN\_CENTER\_LEFT, Y\_ALIGN\_BASELINE\_LEFT, Y\_ALIGN\_BOTTOM\_LEFT, Y\_ALIGN\_TOP\_CENTER, Y\_ALIGN\_CENTER, Y\_ALIGN\_BASELINE\_CENTER, Y\_ALIGN\_BOTTOM\_CENTER, Y\_ALIGN\_TOP\_DECIMAL, Y\_ALIGN\_CENTER\_DECIMAL, Y\_ALIGN\_BASELINE\_DECIMAL, Y\_ALIGN\_BOTTOM\_DECIMAL, Y\_ALIGN\_TOP\_RIGHT, Y\_ALIGN\_CENTER\_RIGHT, Y\_ALIGN\_BASELINE\_RIGHT, Y\_ALIGN\_BOTTOM\_RIGHT.  
**text** the text string to draw

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**displaylayer→get\_display()****YDisplayLayer****displaylayer→display()displaylayer.get\_display()**

---

Gets parent YDisplay.

```
def get_display( )
```

Returns the parent YDisplay object of the current YDisplayLayer.

**Returns :**

an YDisplay object

**displaylayer→get\_displayHeight()**  
**displaylayer→displayHeight()**  
**displaylayer.get\_displayHeight()**

**YDisplayLayer**

Returns the display height, in pixels.

```
def get_displayHeight( )
```

**Returns :**

an integer corresponding to the display height, in pixels On failure, throws an exception or returns Y\_DISPLAYHEIGHT\_INVALID.

**displaylayer→get\_displayWidth()**  
**displaylayer→displayWidth()**  
**displaylayer.get\_displayWidth()**

**YDisplayLayer**

Returns the display width, in pixels.

```
def get_displayWidth( )
```

**Returns :**

an integer corresponding to the display width, in pixels On failure, throws an exception or returns Y\_DISPLAYWIDTH\_INVALID.

**displaylayer→get\_layerHeight()**  
**displaylayer→layerHeight()**  
**displaylayer.get\_layerHeight()**

---

**YDisplayLayer**

Returns the height of the layers to draw on, in pixels.

```
def get_layerHeight( )
```

**Returns :**

an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y\_LAYERHEIGHT\_INVALID.

**displaylayer→get\_layerWidth()**  
**displaylayer→layerWidth()**  
**displaylayer.get\_layerWidth()**

**YDisplayLayer**

Returns the width of the layers to draw on, in pixels.

```
def get_layerWidth( )
```

**Returns :**

an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns Y\_LAYERWIDTH\_INVALID.

**displaylayer→hide()displaylayer.hide()****YDisplayLayer**

Hides the layer.

```
def hide( )
```

The state of the layer is preserved but the layer is not displayed on the screen until the next call to `unhide()`. Hiding the layer can positively affect the drawing speed, since it postpones the rendering until all operations are completed (double-buffering).

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→lineTo()displaylayer.lineTo()****YDisplayLayer**

Draws a line from current drawing pointer position to the specified position.

```
def lineTo( x, y)
```

The specified destination pixel is included in the line. The pointer position is then moved to the end point of the line.

**Parameters :**

- x** the distance from left of layer to the end point of the line, in pixels
- y** the distance from top of layer to the end point of the line, in pixels

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→moveTo()displaylayer.moveTo()****YDisplayLayer**

Moves the drawing pointer of this layer to the specified position.

```
def moveTo( x, y )
```

**Parameters :**

**x** the distance from left of layer, in pixels

**y** the distance from top of layer, in pixels

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→reset()displaylayer.reset()****YDisplayLayer**

Reverts the layer to its initial state (fully transparent, default settings).

```
def reset( )
```

Reinitializes the drawing pointer to the upper left position, and selects the most visible pen color. If you only want to erase the layer content, use the method `clear( )` instead.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→selectColorPen()  
displaylayer.selectColorPen()****YDisplayLayer**

Selects the pen color for all subsequent drawing functions, including text drawing.

```
def selectColorPen( color)
```

The pen color is provided as an RGB value. For grayscale or monochrome displays, the value is automatically converted to the proper range.

**Parameters :**

**color** the desired pen color, as a 24-bit RGB value

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→selectEraser()**  
**displaylayer.selectEraser()****YDisplayLayer**

Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

```
def selectEraser( )
```

Any point drawn using the eraser becomes transparent (as when the layer is empty), showing the other layers beneath it.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→selectFont()displaylayer.selectFont()****YDisplayLayer**

Selects a font to use for the next text drawing functions, by providing the name of the font file.

```
def selectFont( fontname)
```

You can use a built-in font as well as a font file that you have previously uploaded to the device built-in memory. If you experience problems selecting a font file, check the device logs for any error message such as missing font file or bad font file format.

**Parameters :**

**fontname** the font file name

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→selectGrayPen()**  
**displaylayer.selectGrayPen()****YDisplayLayer**

Selects the pen gray level for all subsequent drawing functions, including text drawing.

```
def selectGrayPen( graylevel)
```

The gray level is provided as a number between 0 (black) and 255 (white, or whichever the highest color is). For monochrome displays (without gray levels), any value lower than 128 is rendered as black, and any value equal or above to 128 is non-black.

**Parameters :**

**graylevel** the desired gray level, from 0 to 255

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→setAntialiasingMode()**  
**displaylayer.setAntialiasingMode()****YDisplayLayer**

Enables or disables anti-aliasing for drawing oblique lines and circles.

```
def setAntialiasingMode( mode)
```

Anti-aliasing provides a smoother aspect when looked from far enough, but it can add fuzziness when the display is looked from very close. At the end of the day, it is your personal choice. Anti-aliasing is enabled by default on grayscale and color displays, but you can disable it if you prefer. This setting has no effect on monochrome displays.

**Parameters :**

**mode** true to enable antialiasing, false to disable it.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→setConsoleBackground()**  
**displaylayer.setConsoleBackground()****YDisplayLayer**

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

```
def setConsoleBackground( bgcol)
```

**Parameters :**

**bgcol** the background gray level to use when scrolling (0 = black, 255 = white), or -1 for transparent

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→setConsoleMargins()  
displaylayer.setConsoleMargins()****YDisplayLayer**

Sets up display margins for the `consoleOut` function.

```
def setConsoleMargins( x1, y1, x2, y2)
```

**Parameters :**

- x1** the distance from left of layer to the left margin, in pixels
- y1** the distance from top of layer to the top margin, in pixels
- x2** the distance from left of layer to the right margin, in pixels
- y2** the distance from top of layer to the bottom margin, in pixels

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→setConsoleWordWrap()  
displaylayer.setConsoleWordWrap()****YDisplayLayer**

Sets up the wrapping behaviour used by the `consoleOut` function.

```
def setConsoleWordWrap( wordwrap )
```

**Parameters :**

`wordwrap` true to wrap only between words, false to wrap on the last column anyway.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→setLayerPosition()  
displaylayer.setLayerPosition()****YDisplayLayer**

Sets the position of the layer relative to the display upper left corner.

```
def setLayerPosition( x, y, scrollTime)
```

When smooth scrolling is used, the display offset of the layer is automatically updated during the next milliseconds to animate the move of the layer.

**Parameters :**

**x** the distance from left of display to the upper left corner of the layer

**y** the distance from top of display to the upper left corner of the layer

**scrollTime** number of milliseconds to use for smooth scrolling, or 0 if the scrolling should be immediate.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**displaylayer→unhide()displaylayer.unhide()****YDisplayLayer**

Shows the layer.

```
def unhide( )
```

Shows the layer again after a hide command.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## 3.15. External power supply control interface

Yoctopuce application programming interface allows you to control the power source to use for module functions that require high current. The module can also automatically disconnect the external power when a voltage drop is observed on the external power source (external battery running out of power).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_dualpower.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YDualPower = yoctolib.YDualPower;
php	require_once('yocto_dualpower.php');
cpp	#include "yocto_dualpower.h"
m	#import "yocto_dualpower.h"
pas	uses yocto_dualpower;
vb	yocto_dualpower.vb
cs	yocto_dualpower.cs
java	import com.yoctopuce.YoctoAPI.YDualPower;
py	from yocto_dualpower import *

### Global functions

#### yFindDualPower(func)

Retrieves a dual power control for a given identifier.

#### yFirstDualPower()

Starts the enumeration of dual power controls currently accessible.

### YDualPower methods

#### dualpower→describe()

Returns a short text that describes unambiguously the instance of the power control in the form TYPE (NAME )=SERIAL . FUNCTIONID.

#### dualpower→get\_advertisedValue()

Returns the current value of the power control (no more than 6 characters).

#### dualpower→get\_errorMessage()

Returns the error message of the latest error with the power control.

#### dualpower→get\_errorType()

Returns the numerical error code of the latest error with the power control.

#### dualpower→get\_extVoltage()

Returns the measured voltage on the external power source, in millivolts.

#### dualpower→get\_friendlyName()

Returns a global identifier of the power control in the format MODULE\_NAME . FUNCTION\_NAME.

#### dualpower→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### dualpower→get\_functionId()

Returns the hardware identifier of the power control, without reference to the module.

#### dualpower→get\_hardwareId()

Returns the unique hardware identifier of the power control in the form SERIAL . FUNCTIONID.

#### dualpower→get\_logicalName()

Returns the logical name of the power control.

#### dualpower→get\_module()

Gets the YModule object for the device on which the function is located.

**dualpower→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**dualpower→get\_powerControl()**

Returns the selected power source for module functions that require lots of current.

**dualpower→get\_powerState()**

Returns the current power source for module functions that require lots of current.

**dualpower→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set(userData).

**dualpower→isOnline()**

Checks if the power control is currently reachable, without raising any error.

**dualpower→isOnline\_async(callback, context)**

Checks if the power control is currently reachable, without raising any error (asynchronous version).

**dualpower→load(msValidity)**

Preloads the power control cache with a specified validity duration.

**dualpower→load\_async(msValidity, callback, context)**

Preloads the power control cache with a specified validity duration (asynchronous version).

**dualpower→nextDualPower()**

Continues the enumeration of dual power controls started using yFirstDualPower( ).

**dualpower→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**dualpower→set\_logicalName(newval)**

Changes the logical name of the power control.

**dualpower→set\_powerControl(newval)**

Changes the selected power source for module functions that require lots of current.

**dualpower→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**dualpower→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YDualPower.FindDualPower() yFindDualPower()YDualPower.FindDualPower()

Retrieves a dual power control for a given identifier.

```
def FindDualPower( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the power control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDualPower.isOnline()` to test if the power control is indeed online at a given time. In case of ambiguity when looking for a dual power control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the power control

### Returns :

a `YDualPower` object allowing you to drive the power control.

**YDualPower.FirstDualPower()****yFirstDualPower()YDualPower.FirstDualPower()**

Starts the enumeration of dual power controls currently accessible.

```
def FirstDualPower( )
```

Use the method `YDualPower.nextDualPower( )` to iterate on next dual power controls.

**Returns :**

a pointer to a `YDualPower` object, corresponding to the first dual power control currently online, or a null pointer if there are none.

**dualpower→describe()dualpower.describe()****YDualPower**

Returns a short text that describes unambiguously the instance of the power control in the form  
TYPE ( NAME )=SERIAL . FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the power control (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

**dualpower→get\_advertisedValue()**  
**dualpower→advertisedValue()**  
**dualpower.get\_advertisedValue()**

**YDualPower**

Returns the current value of the power control (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the power control (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**dualpower→get\_errorMessage()**  
**dualpower→errorMessage()**  
**dualpower.get\_errorMessage()**

---

**YDualPower**

Returns the error message of the latest error with the power control.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the power control object

---

**dualpower→get\_errorType()****YDualPower****dualpower→errorType()dualpower.get\_errorType()**

Returns the numerical error code of the latest error with the power control.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the power control object

**dualpower→get\_extVoltage()**

**YDualPower**

**dualpower→extVoltage()dualpower.get\_extVoltage()**

---

Returns the measured voltage on the external power source, in millivolts.

```
def get_extVoltage( )
```

**Returns :**

an integer corresponding to the measured voltage on the external power source, in millivolts

On failure, throws an exception or returns Y\_EXTVOLTAGE\_INVALID.

**dualpower→get\_friendlyName()**  
**dualpower→friendlyName()**  
**dualpower.get\_friendlyName()**

**YDualPower**

Returns a global identifier of the power control in the format MODULE\_NAME . FUNCTION\_NAME.

**def get\_friendlyName( )**

The returned string uses the logical names of the module and of the power control if they are defined, otherwise the serial number of the module and the hardware identifier of the power control (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the power control using logical names (ex: MyCustomName . relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**dualpower→get\_functionDescriptor()**  
**dualpower→functionDescriptor()**  
**dualpower.get\_functionDescriptor()**

**YDualPower**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**dualpower→get\_functionId()****YDualPower****dualpower→functionId()dualpower.get\_functionId()**

Returns the hardware identifier of the power control, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the power control (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**dualpower→get\_hardwareId()**

**YDualPower**

**dualpower→hardwareId()dualpower.get\_hardwareId()**

---

Returns the unique hardware identifier of the power control in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the power control. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the power control (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**dualpower→get\_logicalName()**  
**dualpower→logicalName()**  
**dualpower.get\_logicalName()**

**YDualPower**

Returns the logical name of the power control.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the power control. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**dualpower→get\_module()**

**YDualPower**

**dualpower→module()dualpower.get\_module()**

---

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**dualpower→get\_powerControl()**  
**dualpower→powerControl()**  
**dualpower.get\_powerControl()**

**YDualPower**

Returns the selected power source for module functions that require lots of current.

```
def get_powerControl( )
```

**Returns :**

a value among Y\_POWERCONTROL\_AUTO, Y\_POWERCONTROL\_FROM\_USB, Y\_POWERCONTROL\_FROM\_EXT and Y\_POWERCONTROL\_OFF corresponding to the selected power source for module functions that require lots of current

On failure, throws an exception or returns Y\_POWERCONTROL\_INVALID.

**dualpower→get\_powerState()**  
**dualpower→powerState()**  
**dualpower.get\_powerState()**

**YDualPower**

Returns the current power source for module functions that require lots of current.

```
def get_powerState( )
```

**Returns :**

a value among Y\_POWERSTATE\_OFF, Y\_POWERSTATE\_FROM\_USB and Y\_POWERSTATE\_FROM\_EXT corresponding to the current power source for module functions that require lots of current

On failure, throws an exception or returns Y\_POWERSTATE\_INVALID.

---

**dualpower→get(userData)****YDualPower****dualpower→userData()dualpower.get(userData())**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**dualpower→isOnline()dualpower.isOnline()****YDualPower**

Checks if the power control is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the power control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the power control.

**Returns :**

`true` if the power control can be reached, and `false` otherwise

**dualpower→load()dualpower.load()****YDualPower**

Preloads the power control cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**dualpower→nextDualPower()**  
**dualpower.nextDualPower()**

---

**YDualPower**

Continues the enumeration of dual power controls started using `yFirstDualPower()`.

```
def nextDualPower( )
```

**Returns :**

a pointer to a `YDualPower` object, corresponding to a dual power control currently online, or a null pointer if there are no more dual power controls to enumerate.

**dualpower→registerValueCallback()**  
**dualpower.registerValueCallback()****YDualPower**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**dualpower→set\_logicalName()**  
**dualpower→setLogicalName()**  
**dualpower.set\_logicalName()**

**YDualPower**

Changes the logical name of the power control.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the power control.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**dualpower→set\_powerControl()**  
**dualpower→setPowerControl()**  
**dualpower.set\_powerControl()**

**YDualPower**

Changes the selected power source for module functions that require lots of current.

```
def set_powerControl( newval)
```

**Parameters :**

**newval** a value among Y\_POWERCONTROL\_AUTO, Y\_POWERCONTROL\_FROM\_USB, Y\_POWERCONTROL\_FROM\_EXT and Y\_POWERCONTROL\_OFF corresponding to the selected power source for module functions that require lots of current

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**dualpower→set(userData())**

**YDualPower**

**dualpower→setUserData()dualpower.set(userData())**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.16. Files function interface

The filesystem interface makes it possible to store files on some devices, for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_files.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YFiles = yoctolib.YFiles;
php	require_once('yocto_files.php');
cpp	#include "yocto_files.h"
m	#import "yocto_files.h"
pas	uses yocto_files;
vb	yocto_files.vb
cs	yocto_files.cs
java	import com.yoctopuce.YoctoAPI.YFiles;
py	from yocto_files import *

### Global functions

#### yFindFiles(func)

Retrieves a filesystem for a given identifier.

#### yFirstFiles()

Starts the enumeration of filesystems currently accessible.

### YFiles methods

#### files→describe()

Returns a short text that describes unambiguously the instance of the filesystem in the form  
TYPE ( NAME ) = SERIAL . FUNCTIONID.

#### files→download(pathname)

Downloads the requested file and returns a binary buffer with its content.

#### files→download\_async(pathname, callback, context)

Downloads the requested file and returns a binary buffer with its content.

#### files→format\_fs()

Reinitializes the filesystem to its clean, unfragmented, empty state.

#### files→get\_advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

#### files→get\_errorMessage()

Returns the error message of the latest error with the filesystem.

#### files→get\_errorType()

Returns the numerical error code of the latest error with the filesystem.

#### files→get\_filesCount()

Returns the number of files currently loaded in the filesystem.

#### files→get\_freeSpace()

Returns the free space for uploading new files to the filesystem, in bytes.

#### files→get\_friendlyName()

Returns a global identifier of the filesystem in the format MODULE\_NAME . FUNCTION\_NAME.

#### files→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### files→get\_functionId()

### 3. Reference

Returns the hardware identifier of the filesystem, without reference to the module.

#### **files→get\_hardwareId()**

Returns the unique hardware identifier of the filesystem in the form SERIAL . FUNCTIONID.

#### **files→get\_list(pattern)**

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

#### **files→get\_logicalName()**

Returns the logical name of the filesystem.

#### **files→get\_module()**

Gets the YModule object for the device on which the function is located.

#### **files→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

#### **files→get(userData)**

Returns the value of the userData attribute, as previously stored using method set(userData).

#### **files→isOnline()**

Checks if the filesystem is currently reachable, without raising any error.

#### **files→isOnline\_async(callback, context)**

Checks if the filesystem is currently reachable, without raising any error (asynchronous version).

#### **files→load(msValidity)**

Preloads the filesystem cache with a specified validity duration.

#### **files→load\_async(msValidity, callback, context)**

Preloads the filesystem cache with a specified validity duration (asynchronous version).

#### **files→nextFiles()**

Continues the enumeration of filesystems started using yFirstFiles( ).

#### **files→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

#### **files→remove(pathname)**

Deletes a file, given by its full path name, from the filesystem.

#### **files→set\_logicalName(newval)**

Changes the logical name of the filesystem.

#### **files→set(userData)**

Stores a user context provided as argument in the userData attribute of the function.

#### **files→upload(pathname, content)**

Uploads a file to the filesystem, to the specified full path name.

#### **files→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**YFiles.FindFiles()****YFiles****yFindFiles()YFiles.FindFiles()**

Retrieves a filesystem for a given identifier.

```
def FindFiles( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the filesystem is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YFiles.isOnline()` to test if the filesystem is indeed online at a given time. In case of ambiguity when looking for a filesystem by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the filesystem

**Returns :**

a `YFiles` object allowing you to drive the filesystem.

## **YFiles.FirstFiles()**

**YFiles**

### **yFirstFiles()YFiles.FirstFiles()**

---

Starts the enumeration of filesystems currently accessible.

```
def FirstFiles( )
```

Use the method `YFiles.nextFiles()` to iterate on next filesystems.

**Returns :**

a pointer to a `YFiles` object, corresponding to the first filesystem currently online, or a `null` pointer if there are none.

**files→describe()files.describe()****YFiles**

Returns a short text that describes unambiguously the instance of the filesystem in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the filesystem (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**files→download()files.download()**

YFiles

Downloads the requested file and returns a binary buffer with its content.

```
def download( pathname)
```

**Parameters :**

**pathname** path and name of the file to download

**Returns :**

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

**files→format\_fs()files.format\_fs()****YFiles**

Reinitializes the filesystem to its clean, unfragmented, empty state.

```
def format_fs( )
```

All files previously uploaded are permanently lost.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**files→get\_advertisedValue()** YFiles  
**files→advertisedValue()files.get\_advertisedValue()**

---

Returns the current value of the filesystem (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the filesystem (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

---

**files→get\_errorMessage()****YFiles****files→errorMessage()files.get\_errorMessage()**

---

Returns the error message of the latest error with the filesystem.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the filesystem object

**files→get\_errorType()**  
**files→errorType()files.get\_errorType()**

---

**YFiles**

Returns the numerical error code of the latest error with the filesystem.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the filesystem object

---

**files→get\_filesCount()****YFiles****files→filesCount()files.get\_filesCount()**

Returns the number of files currently loaded in the filesystem.

```
def get_filesCount( )
```

**Returns :**

an integer corresponding to the number of files currently loaded in the filesystem

On failure, throws an exception or returns Y\_FILESCOUNT\_INVALID.

**files→get\_freeSpace()**

**YFiles**

**files→freeSpace()files.get\_freeSpace()**

---

Returns the free space for uploading new files to the filesystem, in bytes.

```
def get_freeSpace( )
```

**Returns :**

an integer corresponding to the free space for uploading new files to the filesystem, in bytes

On failure, throws an exception or returns Y\_FREESPACE\_INVALID.

**files→get\_friendlyName()****YFiles****files→friendlyName()files.get\_friendlyName()**

Returns a global identifier of the filesystem in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the filesystem if they are defined, otherwise the serial number of the module and the hardware identifier of the filesystem (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the filesystem using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

---

<b>files→get_functionDescriptor()</b>	<b>YFiles</b>
<b>files→functionDescriptor()</b>	
<b>files.get_functionDescriptor()</b>	

---

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

---

**files→get\_functionId()****YFiles****files→functionId()files.get\_functionId()**

---

Returns the hardware identifier of the filesystem, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the filesystem (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

---

<b>files→get_hardwareId()</b>	<b>YFiles</b>
<b>files→hardwareId()files.get_hardwareId()</b>	

---

Returns the unique hardware identifier of the filesystem in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the filesystem. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the filesystem (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**files→get\_list()****YFiles****files→list()files.get\_list()**

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

```
def get_list( pattern)
```

**Parameters :**

**pattern** an optional filter pattern, using star and question marks as wildcards. When an empty pattern is provided, all file records are returned.

**Returns :**

a list of YFileRecord objects, containing the file path and name, byte size and 32-bit CRC of the file content.

On failure, throws an exception or returns an empty list.

**files→get\_logicalName()** YFiles  
**files→logicalName()files.get\_logicalName()**

---

Returns the logical name of the filesystem.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the filesystem. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

---

**files→get\_module()****YFiles****files→module()files.get\_module()**

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

**files→get(userData)**

**YFiles**

**files→userData()files.get(userData)**

---

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**files→isOnline()files.isOnline()****YFiles**

Checks if the filesystem is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the filesystem in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the filesystem.

**Returns :**

`true` if the filesystem can be reached, and `false` otherwise

**files→load()files.load()****YFiles**

Preloads the filesystem cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**files→nextFiles()files.nextFiles()****YFiles**

Continues the enumeration of filesystems started using `yFirstFiles()`.

```
def nextFiles( )
```

**Returns :**

a pointer to a `YFiles` object, corresponding to a filesystem currently online, or a `null` pointer if there are no more filesystems to enumerate.

**files→registerValueCallback()  
files.registerValueCallback()****YFiles**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**files→remove()files.remove()****YFiles**

Deletes a file, given by its full path name, from the filesystem.

```
def remove( pathname)
```

Because of filesystem fragmentation, deleting a file may not always free up the whole space used by the file. However, rewriting a file with the same path name will always reuse any space not freed previously. If you need to ensure that no space is taken by previously deleted files, you can use `format_fs` to fully reinitialize the filesystem.

**Parameters :**

**pathname** path and name of the file to remove.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

<b>files→set_logicalName()</b>	<b>YFiles</b>
<b>files→setLogicalName()</b>	

---

Changes the logical name of the filesystem.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the filesystem.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

---

**files→set(userData)****YFiles****files→setUserData()files.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData: Object)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

**files→upload(files.upload())****YFiles**

Uploads a file to the filesystem, to the specified full path name.

```
def upload( pathname, content)
```

If a file already exists with the same path name, its content is overwritten.

**Parameters :**

**pathname** path and name of the new file to create

**content** binary buffer with the content to set

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## 3.17. GenericSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_geneticsensor.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YGenericSensor = yoctolib.YGenericSensor;
php	require_once('yocto_geneticsensor.php');
cpp	#include "yocto_geneticsensor.h"
m	#import "yocto_geneticsensor.h"
pas	uses yocto_geneticsensor;
vb	yocto_geneticsensor.vb
cs	yocto_geneticsensor.cs
java	import com.yoctopuce.YoctoAPI.YGenericSensor;
py	from yocto_geneticsensor import *

### Global functions

#### yFindGenericSensor(func)

Retrieves a generic sensor for a given identifier.

#### yFirstGenericSensor()

Starts the enumeration of generic sensors currently accessible.

### YGenericSensor methods

#### geneticsensor→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### geneticsensor→describe()

Returns a short text that describes unambiguously the instance of the generic sensor in the form TYPE (NAME )=SERIAL . FUNCTIONID.

#### geneticsensor→get\_advertisedValue()

Returns the current value of the generic sensor (no more than 6 characters).

#### geneticsensor→get\_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### geneticsensor→get\_currentValue()

Returns the current measured value.

#### geneticsensor→get\_errorMessage()

Returns the error message of the latest error with the generic sensor.

#### geneticsensor→get\_errorType()

Returns the numerical error code of the latest error with the generic sensor.

#### geneticsensor→get\_friendlyName()

Returns a global identifier of the generic sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### geneticsensor→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### geneticsensor→get\_functionId()

Returns the hardware identifier of the generic sensor, without reference to the module.

#### geneticsensor→get\_hardwareId()

Returns the unique hardware identifier of the generic sensor in the form SERIAL . FUNCTIONID.

<b>genericsensor→get_highestValue()</b>	Returns the maximal value observed for the measure since the device was started.
<b>genericsensor→get_logFrequency()</b>	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>genericsensor→get_logicalName()</b>	Returns the logical name of the generic sensor.
<b>genericsensor→get_lowestValue()</b>	Returns the minimal value observed for the measure since the device was started.
<b>genericsensor→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>genericsensor→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>genericsensor→get_recordedData(startTime, endTime)</b>	Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>genericsensor→get_reportFrequency()</b>	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>genericsensor→get_resolution()</b>	Returns the resolution of the measured values.
<b>genericsensor→get_signalRange()</b>	Returns the electric signal range used by the sensor.
<b>genericsensor→get_signalUnit()</b>	Returns the measuring unit of the electrical signal used by the sensor.
<b>genericsensor→get_signalValue()</b>	Returns the measured value of the electrical signal used by the sensor.
<b>genericsensor→get_unit()</b>	Returns the measuring unit for the measure.
<b>genericsensor→get(userData)</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>genericsensor→get_valueRange()</b>	Returns the physical value range measured by the sensor.
<b>genericsensor→isOnline()</b>	Checks if the generic sensor is currently reachable, without raising any error.
<b>genericsensor→isOnline_async(callback, context)</b>	Checks if the generic sensor is currently reachable, without raising any error (asynchronous version).
<b>genericsensor→load(msValidity)</b>	Preloads the generic sensor cache with a specified validity duration.
<b>genericsensor→loadCalibrationPoints(rawValues, refValues)</b>	Retrieves error correction data points previously entered using the method calibrateFromPoints.
<b>genericsensor→load_async(msValidity, callback, context)</b>	Preloads the generic sensor cache with a specified validity duration (asynchronous version).
<b>genericsensor→nextGenericSensor()</b>	Continues the enumeration of generic sensors started using yFirstGenericSensor( ).
<b>genericsensor→registerTimedReportCallback(callback)</b>	Registers the callback function that is invoked on every periodic timed notification.

**genericsensor→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**genericsensor→set\_highestValue(newval)**

Changes the recorded maximal value observed.

**genericsensor→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**genericsensor→set\_logicalName(newval)**

Changes the logical name of the generic sensor.

**genericsensor→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**genericsensor→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**genericsensor→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**genericsensor→set\_signalRange(newval)**

Changes the electric signal range used by the sensor.

**genericsensor→set\_unit(newval)**

Changes the measuring unit for the measured value.

**genericsensor→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**genericsensor→set\_valueRange(newval)**

Changes the physical value range measured by the sensor.

**genericsensor→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**YGenericSensor.FindGenericSensor()**  
**yFindGenericSensor()**  
**YGenericSensor.FindGenericSensor()****YGenericSensor**

Retrieves a generic sensor for a given identifier.

```
def FindGenericSensor( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the generic sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGenericSensor.isOnline()` to test if the generic sensor is indeed online at a given time. In case of ambiguity when looking for a generic sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the generic sensor

**Returns :**

a `YGenericSensor` object allowing you to drive the generic sensor.

**YGenericSensor.FirstGenericSensor()**  
**yFirstGenericSensor()**  
**YGenericSensor.FirstGenericSensor()****YGenericSensor**

Starts the enumeration of generic sensors currently accessible.

```
def FirstGenericSensor( )
```

Use the method `YGenericSensor.nextGenericSensor()` to iterate on next generic sensors.

**Returns :**

a pointer to a `YGenericSensor` object, corresponding to the first generic sensor currently online, or a null pointer if there are none.

**genericsensor→calibrateFromPoints()**  
**genericsensor.calibrateFromPoints()****YGenericSensor**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**genericsensor→describe()genericsensor.describe()****YGenericSensor**

Returns a short text that describes unambiguously the instance of the generic sensor in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the generic sensor (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**genericsensor→get\_advertisedValue()**  
**genericsensor→advertisedValue()**  
**genericsensor.get\_advertisedValue()**

---

**YGenericSensor**

Returns the current value of the generic sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the generic sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**genericsensor→get\_currentRawValue()**  
**genericsensor→currentRawValue()**  
**genericsensor.get\_currentRawValue()**

**YGenericSensor**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**genericsensor→get\_currentValue()**  
**genericsensor→currentValue()**  
**genericsensor.get\_currentValue()**

---

**YGenericSensor**

Returns the current measured value.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current measured value

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

---

**genericsensor→get\_errorMessage()**  
**genericsensor→errorMessage()**  
**genericsensor.get\_errorMessage()**

**YGenericSensor**

Returns the error message of the latest error with the generic sensor.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the generic sensor object

**genericsensor→get\_errorType()**  
**genericsensor→errorType()**  
**genericsensor.get\_errorType()**

**YGenericSensor**

---

Returns the numerical error code of the latest error with the generic sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the generic sensor object

**genericsensor→get\_friendlyName()**  
**genericsensor→friendlyName()**  
**genericsensor.get\_friendlyName()**

**YGenericSensor**

Returns a global identifier of the generic sensor in the format MODULE\_NAME . FUNCTION\_NAME.

**def get\_friendlyName( )**

The returned string uses the logical names of the module and of the generic sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the generic sensor (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the generic sensor using logical names (ex: MyCustomName . relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**genericsensor→get\_functionDescriptor()**  
**genericsensor→functionDescriptor()**  
**genericsensor.get\_functionDescriptor()**

**YGenericSensor**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**genicsensor→get\_functionId()**  
**genicsensor→functionId()**  
**genicsensor.get\_functionId()**

**YGenericSensor**

Returns the hardware identifier of the generic sensor, without reference to the module.

```
def get_functionId( )
```

For example relay1

**Returns :**

a string that identifies the generic sensor (ex: relay1) On failure, throws an exception or returns Y\_FUNCTIONID\_INVALID.

**genericsensor→get\_hardwareId()**  
**genericsensor→hardwareId()**  
**genericsensor.get\_hardwareId()**

**YGenericSensor**

Returns the unique hardware identifier of the generic sensor in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the generic sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the generic sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**genericsensor→get\_highestValue()**  
**genericsensor→highestValue()**  
**genericsensor.get\_highestValue()**

**YGenericSensor**

Returns the maximal value observed for the measure since the device was started.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**genericsensor→get\_logFrequency()**  
**genericsensor→logFrequency()**  
**genericsensor.get\_logFrequency()**

**YGenericSensor**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**genericsensor→get\_logicalName()**  
**genericsensor→logicalName()**  
**genericsensor.get\_logicalName()**

**YGenericSensor**

Returns the logical name of the generic sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the generic sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**genericsensor→get\_lowestValue()**  
**genericsensor→lowestValue()**  
**genericsensor.get\_lowestValue()**

**YGenericSensor**

Returns the minimal value observed for the measure since the device was started.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**genericsensor→get\_module()**  
**genericsensor→module()**  
**genericsensor.get\_module()**

**YGenericSensor**

Gets the **YModule** object for the device on which the function is located.

**def get\_module( )**

If the function cannot be located on any module, the returned instance of **YModule** is not shown as online.

**Returns :**

an instance of **YModule**

**genericsensor→get\_recordedData()**  
**genericsensor→recordedData()**  
**genericsensor.get\_recordedData()**

**YGenericSensor**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime )
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**genericsensor→get\_reportFrequency()**  
**genericsensor→reportFrequency()**  
**genericsensor.get\_reportFrequency()**

**YGenericSensor**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**genericsensor→get\_resolution()**  
**genericsensor→resolution()**  
**genericsensor.get\_resolution()**

---

**YGenericSensor**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**genericsensor→get\_signalRange()**  
**genericsensor→signalRange()**  
**genericsensor.get\_signalRange()**

**YGenericSensor**

Returns the electric signal range used by the sensor.

```
def get_signalRange( )
```

**Returns :**

a string corresponding to the electric signal range used by the sensor

On failure, throws an exception or returns Y\_SIGNALRANGE\_INVALID.

**genericsensor→get\_signalUnit()**  
**genericsensor→signalUnit()**  
**genericsensor.get\_signalUnit()**

---

**YGenericSensor**

Returns the measuring unit of the electrical signal used by the sensor.

```
def get_signalUnit( )
```

**Returns :**

a string corresponding to the measuring unit of the electrical signal used by the sensor

On failure, throws an exception or returns Y\_SIGNALUNIT\_INVALID.

**genericsensor→get\_signalValue()**  
**genericsensor→signalValue()**  
**genericsensor.get\_signalValue()**

**YGenericSensor**

Returns the measured value of the electrical signal used by the sensor.

```
def get_signalValue( )
```

**Returns :**

a floating point number corresponding to the measured value of the electrical signal used by the sensor

On failure, throws an exception or returns Y\_SIGNALVALUE\_INVALID.

**genericsensor→get\_unit()**

**YGenericSensor**

**genericsensor→unit()genericsensor.get\_unit()**

---

Returns the measuring unit for the measure.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns Y\_UNIT\_INVALID.

---

**genericsensor→get(userData)**  
**genericsensor→userData()**  
**genericsensor.get(userData)**

**YGenericSensor**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

`def get(userData) :`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**genericsensor→get\_valueRange()**  
**genericsensor→valueRange()**  
**genericsensor.get\_valueRange()**

**YGenericSensor**

---

Returns the physical value range measured by the sensor.

```
def get_valueRange( )
```

**Returns :**

a string corresponding to the physical value range measured by the sensor

On failure, throws an exception or returns Y\_VALUERANGE\_INVALID.

**genericsensor→isOnline()genericsensor.isOnline()****YGenericSensor**

Checks if the generic sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the generic sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the generic sensor.

**Returns :**

`true` if the generic sensor can be reached, and `false` otherwise

**genericsensor→load()genericsensor.load()****YGenericSensor**

Preloads the generic sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**genericsensor→loadCalibrationPoints()  
genericsensor.loadCalibrationPoints()****YGenericSensor**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**genericsensor→nextGenericSensor()**  
**genericsensor.nextGenericSensor()**

---

**YGenericSensor**

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

```
def nextGenericSensor( )
```

**Returns :**

a pointer to a `YGenericSensor` object, corresponding to a generic sensor currently online, or a `null` pointer if there are no more generic sensors to enumerate.

**genericsensor→registerTimedReportCallback()**  
**genericsensor.registerTimedReportCallback()****YGenericSensor**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**genericsensor→registerValueCallback()  
genericsensor.registerValueCallback()****YGenericSensor**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**genericsensor→set\_highestValue()**  
**genericsensor→setHighestValue()**  
**genericsensor.set\_highestValue()**

**YGenericSensor**

Changes the recorded maximal value observed.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**genericsensor→set\_logFrequency()**  
**genericsensor→setLogFrequency()**  
**genericsensor.set\_logFrequency()**

**YGenericSensor**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**genericsensor→set\_logicalName()**  
**genericsensor→setLogicalName()**  
**genericsensor.set\_logicalName()**

**YGenericSensor**

Changes the logical name of the generic sensor.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the generic sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**genericsensor→set\_lowestValue()**  
**genericsensor→setLowestValue()**  
**genericsensor.set\_lowestValue()**

**YGenericSensor**

---

Changes the recorded minimal value observed.

```
def set_lowestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**genericsensor→set\_reportFrequency()**  
**genericsensor→setReportFrequency()**  
**genericsensor.set\_reportFrequency()**

**YGenericSensor**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**genericsensor→set\_resolution()**  
**genericsensor→setResolution()**  
**genericsensor.set\_resolution()**

**YGenericSensor**

Changes the resolution of the measured physical values.

```
def set_resolution( newval )
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**genericsensor→set\_signalRange()**  
**genericsensor→setSignalRange()**  
**genericsensor.set\_signalRange()**

**YGenericSensor**

Changes the electric signal range used by the sensor.

```
def set_signalRange( newval)
```

**Parameters :**

**newval** a string corresponding to the electric signal range used by the sensor

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**genericsensor→set\_unit()**

**YGenericSensor**

**genericsensor→setUnit()genericsensor.set\_unit()**

---

Changes the measuring unit for the measured value.

```
def set_unit( newval)
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the measuring unit for the measured value

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**genericsensor→set(userData)**  
**genericsensor→setUserData()**  
**genericsensor.set(userData)**

**YGenericSensor**

Stores a user context provided as argument in the userData attribute of the function.

```
def setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

**genericsensor→set\_valueRange()**  
**genericsensor→setValueRange()**  
**genericsensor.set\_valueRange()**

---

**YGenericSensor**

Changes the physical value range measured by the sensor.

```
def set_valueRange( newval)
```

The range change may have a side effect on the display resolution, as it may be adapted automatically.

**Parameters :**

**newval** a string corresponding to the physical value range measured by the sensor

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## 3.18. Gyroscope function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_gyro.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YGyro = yoctolib.YGyro;
php	require_once('yocto_gyro.php');
cpp	#include "yocto_gyro.h"
m	#import "yocto_gyro.h"
pas	uses yocto_gyro;
vb	yocto_gyro.vb
cs	yocto_gyro.cs
java	import com.yoctopuce.YoctoAPI.YGyro;
py	from yocto_gyro import *

### Global functions

#### **yocto\_gyro(func)**

Retrieves a gyroscope for a given identifier.

#### **yFirstGyro()**

Starts the enumeration of gyroscopes currently accessible.

### YGyro methods

#### **gyro->calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### **gyro->describe()**

Returns a short text that describes unambiguously the instance of the gyroscope in the form TYPE (NAME )=SERIAL . FUNCTIONID.

#### **gyro->get\_advertisedValue()**

Returns the current value of the gyroscope (no more than 6 characters).

#### **gyro->get\_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### **gyro->get\_currentValue()**

Returns the current value of the angular velocity.

#### **gyro->get\_errorMessage()**

Returns the error message of the latest error with the gyroscope.

#### **gyro->get\_errorType()**

Returns the numerical error code of the latest error with the gyroscope.

#### **gyro->get\_friendlyName()**

Returns a global identifier of the gyroscope in the format MODULE\_NAME . FUNCTION\_NAME.

#### **gyro->get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### **gyro->get\_functionId()**

Returns the hardware identifier of the gyroscope, without reference to the module.

#### **gyro->get\_hardwareId()**

Returns the unique hardware identifier of the gyroscope in the form SERIAL . FUNCTIONID.

<b>gyro→get_heading()</b>	Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.
<b>gyro→get_highestValue()</b>	Returns the maximal value observed for the angular velocity since the device was started.
<b>gyro→get_logFrequency()</b>	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>gyro→get_logicalName()</b>	Returns the logical name of the gyroscope.
<b>gyro→get_lowestValue()</b>	Returns the minimal value observed for the angular velocity since the device was started.
<b>gyro→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>gyro→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>gyro→get_pitch()</b>	Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.
<b>gyro→get_quaternionW()</b>	Returns the w component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.
<b>gyro→get_quaternionX()</b>	Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.
<b>gyro→get_quaternionY()</b>	Returns the y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.
<b>gyro→get_quaternionZ()</b>	Returns the z component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.
<b>gyro→get_recordedData(startTime, endTime)</b>	Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>gyro→get_reportFrequency()</b>	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>gyro→get_resolution()</b>	Returns the resolution of the measured values.
<b>gyro→get_roll()</b>	Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.
<b>gyro→get_unit()</b>	Returns the measuring unit for the angular velocity.
<b>gyro→get_userData()</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>gyro→get_xValue()</b>	Returns the angular velocity around the X axis of the device, as a floating point number.
<b>gyro→get_yValue()</b>	

Returns the angular velocity around the Y axis of the device, as a floating point number.

**gyro→get\_zValue()**

Returns the angular velocity around the Z axis of the device, as a floating point number.

**gyro→isOnline()**

Checks if the gyroscope is currently reachable, without raising any error.

**gyro→isOnline\_async(callback, context)**

Checks if the gyroscope is currently reachable, without raising any error (asynchronous version).

**gyro→load(msValidity)**

Preloads the gyroscope cache with a specified validity duration.

**gyro→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

**gyro→load\_async(msValidity, callback, context)**

Preloads the gyroscope cache with a specified validity duration (asynchronous version).

**gyro→nextGyro()**

Continues the enumeration of gyroscopes started using yFirstGyro( ).

**gyro→registerAnglesCallback(callback)**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

**gyro→registerQuaternionCallback(callback)**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

**gyro→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**gyro→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**gyro→set\_highestValue(newval)**

Changes the recorded maximal value observed.

**gyro→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**gyro→set\_logicalName(newval)**

Changes the logical name of the gyroscope.

**gyro→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**gyro→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**gyro→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**gyro→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**gyro→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YGyro.FindGyro() yFindGyro()YGyro.FindGyro()

YGyro

Retrieves a gyroscope for a given identifier.

```
def FindGyro( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the gyroscope is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGyro.isOnline()` to test if the gyroscope is indeed online at a given time. In case of ambiguity when looking for a gyroscope by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the gyroscope

### Returns :

a `YGyro` object allowing you to drive the gyroscope.

**YGyro.FirstGyro()****YGyro****yFirstGyro()YGyro.FirstGyro()**

Starts the enumeration of gyroscopes currently accessible.

```
def FirstGyro( )
```

Use the method `YGyro.nextGyro()` to iterate on next gyroscopes.

**Returns :**

a pointer to a `YGyro` object, corresponding to the first gyro currently online, or a `null` pointer if there are none.

**gyro→calibrateFromPoints()  
gyro.calibrateFromPoints()****YGyro**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**gyro→describe()gyro.describe()****YGyro**

Returns a short text that describes unambiguously the instance of the gyroscope in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the gyroscope (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**gyro→get\_advertisedValue()**

**YGyro**

**gyro→advertisedValue()gyro.get\_advertisedValue()**

---

Returns the current value of the gyroscope (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the gyroscope (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

---

**gyro→get\_currentRawValue()****YGyro****gyro→currentRawValue()gyro.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**gyro→get\_currentValue()** **YGyro**  
**gyro→currentValue()gyro.get\_currentValue()**

---

Returns the current value of the angular velocity.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the angular velocity

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

---

**gyro→get\_errorMessage()****YGyro****gyro→errorMessage()gyro.get\_errorMessage()**

---

Returns the error message of the latest error with the gyroscope.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the gyroscope object

**gyro→get\_errorType()**

**YGyro**

**gyro→errorType()gyro.get\_errorType()**

---

Returns the numerical error code of the latest error with the gyroscope.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the gyroscope object

**gyro→get\_friendlyName()****YGyro****gyro→friendlyName()gyro.get\_friendlyName()**

Returns a global identifier of the gyroscope in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the gyroscope if they are defined, otherwise the serial number of the module and the hardware identifier of the gyroscope (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the gyroscope using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

---

<b>gyro→get_functionDescriptor()</b>	<b>YGyro</b>
<b>gyro→functionDescriptor()</b>	
<b>gyro.get_functionDescriptor()</b>	

---

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

---

**gyro→get\_functionId()****YGyro****gyro→functionId()gyro.get\_functionId()**

---

Returns the hardware identifier of the gyroscope, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the gyroscope (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

---

<b>gyro→get_hardwareId()</b>	<b>YGyro</b>
<b>gyro→hardwareId()gyro.get_hardwareId()</b>	

---

Returns the unique hardware identifier of the gyroscope in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the gyroscope. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the gyroscope (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

---

**gyro→get\_heading()****YGyro****gyro→heading()gyro.get\_heading()**

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
def get_heading( )
```

The axis corresponding to the heading can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

**Returns :**

a floating-point number corresponding to heading in degrees, between 0 and 360.

**gyro→get\_highestValue()**

**YGyro**

**gyro→highestValue()gyro.get\_highestValue()**

---

Returns the maximal value observed for the angular velocity since the device was started.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the angular velocity since the device was started

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**gyro→get\_logFrequency()****YGyro****gyro→logFrequency()gyro.get\_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

<b>gyro→get_logicalName()</b>	<b>YGyro</b>
<b>gyro→logicalName()gyro.get_logicalName()</b>	

---

Returns the logical name of the gyroscope.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the gyroscope. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**gyro→get\_lowestValue()****YGyro****gyro→lowestValue()gyro.get\_lowestValue()**

Returns the minimal value observed for the angular velocity since the device was started.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the angular velocity since the device was started

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

---

<b>gyro→get_module()</b>	<b>YGyro</b>
<b>gyro→module()gyro.get_module()</b>	

---

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**gyro→get\_pitch()****YGyro****gyro→pitch()gyro.get\_pitch()**

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
def get_pitch( )
```

The axis corresponding to the pitch angle can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

**Returns :**

a floating-point number corresponding to pitch angle in degrees, between -90 and +90.

**gyro→get\_quaternionW()****YGyro****gyro→quaternionW()gyro.get\_quaternionW()**

Returns the w component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
def get_quaternionW( )
```

**Returns :**

a floating-point number corresponding to the w component of the quaternion.

---

**gyro→get\_quaternionX()****YGyro****gyro→quaternionX()gyro.get\_quaternionX()**

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
def get_quaternionX( )
```

The x component is mostly correlated with rotations on the roll axis.

**Returns :**

a floating-point number corresponding to the x component of the quaternion.

---

<b>gyro→get_quaternionY()</b>	<b>YGyro</b>
<b>gyro→quaternionY()gyro.get_quaternionY()</b>	

---

Returns the *y* component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
def get_quaternionY( )
```

The *y* component is mostly correlated with rotations on the pitch axis.

**Returns :**

a floating-point number corresponding to the *y* component of the quaternion.

---

**gyro→get\_quaternionZ()****YGyro****gyro→quaternionZ()gyro.get\_quaternionZ()**

Returns the *x* component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
def get_quaternionZ( )
```

The *x* component is mostly correlated with changes of heading.

**Returns :**

a floating-point number corresponding to the *z* component of the quaternion.

---

<b>gyro→get_recordedData()</b>	<b>YGyro</b>
<b>gyro→recordedData()gyro.get_recordedData()</b>	

---

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime )
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

#### Parameters :

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

#### Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**gyro→get\_reportFrequency()****YGyro****gyro→reportFrequency()gyro.get\_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**gyro→get\_resolution()** YGyro  
**gyro→resolution()gyro.get\_resolution()**

---

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**gyro→get\_roll()****YGyro****gyro→roll()gyro.get\_roll()**

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
def get_roll( )
```

The axis corresponding to the roll angle can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

**Returns :**

a floating-point number corresponding to roll angle in degrees, between -180 and +180.

<b>gyro→get_unit()</b>	<b>YGyro</b>
<b>gyro→unit()gyro.get_unit()</b>	

---

Returns the measuring unit for the angular velocity.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the angular velocity

On failure, throws an exception or returns Y\_UNIT\_INVALID.

---

**gyro→get(userData)****YGyro****gyro→userData()gyro.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**gyro→get\_xValue()**

**YGyro**

**gyro→xValue()gyro.get\_xValue()**

---

Returns the angular velocity around the X axis of the device, as a floating point number.

```
def get_xValue( )
```

**Returns :**

a floating point number corresponding to the angular velocity around the X axis of the device, as a floating point number

On failure, throws an exception or returns Y\_XVALUE\_INVALID.

---

**gyro→get\_yValue()****YGyro****gyro→yValue()gyro.get\_yValue()**

Returns the angular velocity around the Y axis of the device, as a floating point number.

```
def get_yValue( )
```

**Returns :**

a floating point number corresponding to the angular velocity around the Y axis of the device, as a floating point number

On failure, throws an exception or returns Y\_YVALUE\_INVALID.

**gyro→get\_zValue()**

**YGyro**

**gyro→zValue()gyro.get\_zValue()**

---

Returns the angular velocity around the Z axis of the device, as a floating point number.

```
def get_zValue( )
```

**Returns :**

a floating point number corresponding to the angular velocity around the Z axis of the device, as a floating point number

On failure, throws an exception or returns Y\_ZVALUE\_INVALID.

**gyro→isOnline()gyro.isOnline()****YGyro**

Checks if the gyroscope is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the gyroscope in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the gyroscope.

**Returns :**

`true` if the gyroscope can be reached, and `false` otherwise

**gyro→load()gyro.load()****YGyro**

Preloads the gyroscope cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**gyro→loadCalibrationPoints()**  
**gyro.loadCalibrationPoints()****YGyro**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## **gyro→nextGyro()gyro.nextGyro()**

**YGyro**

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

```
def nextGyro( )
```

**Returns :**

a pointer to a `YGyro` object, corresponding to a gyroscope currently online, or a null pointer if there are no more gyroscopes to enumerate.

**gyro→registerAnglesCallback()  
gyro.registerAnglesCallback()****YGyro**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```
def registerAnglesCallback( callback)
```

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to invoke, or a null pointer. The callback function should take four arguments: the YGyro object of the turning device, and the floating point values of the three angles roll, pitch and heading in degrees (as floating-point numbers).

**gyro→registerQuaternionCallback()  
gyro.registerQuaternionCallback()****YGyro**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```
def registerQuaternionCallback( callback)
```

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to invoke, or a null pointer. The callback function should take five arguments: the YGyro object of the turning device, and the floating point values of the four components w, x, y and z (as floating-point numbers).

**gyro→registerTimedReportCallback()  
gyro.registerTimedReportCallback()****YGyro**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**gyro→registerValueCallback()**  
**gyro.registerValueCallback()****YGyro**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**gyro→set\_highestValue()****YGyro****gyro→setHighestValue()gyro.set\_highestValue()**

Changes the recorded maximal value observed.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

<b>gyro→set_logFrequency()</b>	<b>YGyro</b>
<b>gyro→setLogFrequency()gyro.set_logFrequency()</b>	

---

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**gyro→set\_logicalName()****YGyro****gyro→setLogicalName()gyro.set\_logicalName()**

Changes the logical name of the gyroscope.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the gyroscope.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

---

<b>gyro→set_lowestValue()</b>	<b>YGyro</b>
<b>gyro→setLowestValue()</b> gyro.set_lowestValue()	

---

Changes the recorded minimal value observed.

```
def set_lowestValue( newval )
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**gyro→set\_reportFrequency()**  
**gyro→setReportFrequency()**  
**gyro.set\_reportFrequency()**

**YGyro**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

<b>gyro→set_resolution()</b>	<b>YGyro</b>
<b>gyro→setResolution()gyro.set_resolution()</b>	

---

Changes the resolution of the measured physical values.

```
def set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**gyro→set(userData)****YGyro****gyro→setUserData()gyro.set(userData)**

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData: Any)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.19. Yocto-hub port interface

YHubPort objects provide control over the power supply for every YoctoHub port and provide information about the device connected to it. The logical name of a YHubPort is always automatically set to the unique serial number of the Yoctopuce device connected to it.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_hubport.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YHubPort = yoctolib.YHubPort;
php	require_once('yocto_hubport.php');
cpp	#include "yocto_hubport.h"
m	#import "yocto_hubport.h"
pas	uses yocto_hubport;
vb	yocto_hubport.vb
cs	yocto_hubport.cs
java	import com.yoctopuce.YoctoAPI.YHubPort;
py	from yocto_hubport import *

### Global functions

#### yFindHubPort(func)

Retrieves a Yocto-hub port for a given identifier.

#### yFirstHubPort()

Starts the enumeration of Yocto-hub ports currently accessible.

### YHubPort methods

#### hubport→describe()

Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form TYPE (NAME )=SERIAL . FUNCTIONID.

#### hubport→get\_advertisedValue()

Returns the current value of the Yocto-hub port (no more than 6 characters).

#### hubport→get\_baudRate()

Returns the current baud rate used by this Yocto-hub port, in kbps.

#### hubport→get\_enabled()

Returns true if the Yocto-hub port is powered, false otherwise.

#### hubport→get\_errorMessage()

Returns the error message of the latest error with the Yocto-hub port.

#### hubport→get\_errorType()

Returns the numerical error code of the latest error with the Yocto-hub port.

#### hubport→get\_friendlyName()

Returns a global identifier of the Yocto-hub port in the format MODULE\_NAME . FUNCTION\_NAME.

#### hubport→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### hubport→get\_functionId()

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

#### hubport→get\_hardwareId()

Returns the unique hardware identifier of the Yocto-hub port in the form SERIAL . FUNCTIONID.

#### hubport→get\_logicalName()

Returns the logical name of the Yocto-hub port.

<b>hubport→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>hubport→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>hubport→get_portState()</b>	Returns the current state of the Yocto-hub port.
<b>hubport→get_userData()</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>hubport→isOnline()</b>	Checks if the Yocto-hub port is currently reachable, without raising any error.
<b>hubport→isOnline_async(callback, context)</b>	Checks if the Yocto-hub port is currently reachable, without raising any error (asynchronous version).
<b>hubport→load(msValidity)</b>	Preloads the Yocto-hub port cache with a specified validity duration.
<b>hubport→load_async(msValidity, callback, context)</b>	Preloads the Yocto-hub port cache with a specified validity duration (asynchronous version).
<b>hubport→nextHubPort()</b>	Continues the enumeration of Yocto-hub ports started using yFirstHubPort( ).
<b>hubport→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>hubport→set_enabled(newval)</b>	Changes the activation of the Yocto-hub port.
<b>hubport→set_logicalName(newval)</b>	Changes the logical name of the Yocto-hub port.
<b>hubport→set_userData(data)</b>	Stores a user context provided as argument in the userData attribute of the function.
<b>hubport→wait_async(callback, context)</b>	Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**YHubPort.FindHubPort()****YHubPort****yFindHubPort()YHubPort.FindHubPort()**

Retrieves a Yocto-hub port for a given identifier.

```
def FindHubPort( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the Yocto-hub port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YHubPort.isOnline()` to test if the Yocto-hub port is indeed online at a given time. In case of ambiguity when looking for a Yocto-hub port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the Yocto-hub port

**Returns :**

a `YHubPort` object allowing you to drive the Yocto-hub port.

**YHubPort.FirstHubPort()****YHubPort****yFirstHubPort()YHubPort.FirstHubPort()**

Starts the enumeration of Yocto-hub ports currently accessible.

```
def FirstHubPort( )
```

Use the method `YHubPort.nextHubPort()` to iterate on next Yocto-hub ports.

**Returns :**

a pointer to a `YHubPort` object, corresponding to the first Yocto-hub port currently online, or a `null` pointer if there are none.

**hubport→describe()hubport.describe()****YHubPort**

Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form  
TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the Yocto-hub port (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**hubport→get\_advertisedValue()**  
**hubport→advertisedValue()**  
**hubport.get\_advertisedValue()**

**YHubPort**

Returns the current value of the Yocto-hub port (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the Yocto-hub port (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

<b>hubport→get_baudRate()</b>	<b>YHubPort</b>
<b>hubport→baudRate()hubport.get_baudRate()</b>	

---

Returns the current baud rate used by this Yocto-hub port, in kbps.

```
def get_baudRate( )
```

The default value is 1000 kbps, but a slower rate may be used if communication problems are encountered.

**Returns :**

an integer corresponding to the current baud rate used by this Yocto-hub port, in kbps

On failure, throws an exception or returns Y\_BAUDRATE\_INVALID.

**hubport→get\_enabled()****YHubPort****hubport→enabled()hubport.get\_enabled()**

Returns true if the Yocto-hub port is powered, false otherwise.

```
def get_enabled( )
```

**Returns :**

either Y\_ENABLED\_FALSE or Y\_ENABLED\_TRUE, according to true if the Yocto-hub port is powered, false otherwise

On failure, throws an exception or returns Y\_ENABLED\_INVALID.

**hubport→get\_errorMessage()** YHubPort  
**hubport→errorMessage()hubport.get\_errorMessage()**

---

Returns the error message of the latest error with the Yocto-hub port.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the Yocto-hub port object

---

**hubport→get\_errorType()****YHubPort****hubport→errorType()hubport.get\_errorType()**

Returns the numerical error code of the latest error with the Yocto-hub port.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the Yocto-hub port object

---

<b>hubport→get_friendlyName()</b>	<b>YHubPort</b>
<b>hubport→friendlyName()hubport.get_friendlyName()</b>	

---

Returns a global identifier of the Yocto-hub port in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the Yocto-hub port if they are defined, otherwise the serial number of the module and the hardware identifier of the Yocto-hub port (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the Yocto-hub port using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**hubport→get\_functionDescriptor()**  
**hubport→functionDescriptor()**  
**hubport.get\_functionDescriptor()**

**YHubPort**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**hubport→get\_functionId()**

**YHubPort**

**hubport→functionId()hubport.get\_functionId()**

---

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the Yocto-hub port (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**hubport→get\_hardwareId()****YHubPort****hubport→hardwareId()hubport.get\_hardwareId()**

Returns the unique hardware identifier of the Yocto-hub port in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the Yocto-hub port. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the Yocto-hub port (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**hubport→get\_logicalName()**

**YHubPort**

**hubport→logicalName()hubport.get\_logicalName()**

---

Returns the logical name of the Yocto-hub port.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the Yocto-hub port. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**hubport→get\_module()****YHubPort****hubport→module()hubport.get\_module()**

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

**hubport→get\_portState()** YHubPort  
**hubport→portState()hubport.get\_portState()**

---

Returns the current state of the Yocto-hub port.

```
def get_portState( )
```

**Returns :**

a value among Y\_PORTSTATE\_OFF, Y\_PORTSTATE\_OVRLD, Y\_PORTSTATE\_ON, Y\_PORTSTATE\_RUN and Y\_PORTSTATE\_PROG corresponding to the current state of the Yocto-hub port

On failure, throws an exception or returns Y\_PORTSTATE\_INVALID.

**hubport→get(userData)****YHubPort****hubport→userData()hubport.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**hubport→isOnline()hubport.isOnline()****YHubPort**

Checks if the Yocto-hub port is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the Yocto-hub port in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Yocto-hub port.

**Returns :**

true if the Yocto-hub port can be reached, and false otherwise

**hubport→load()hubport.load()****YHubPort**

Preloads the Yocto-hub port cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## **hubport→nextHubPort()hubport.nextHubPort()**

**YHubPort**

Continues the enumeration of Yocto-hub ports started using `yFirstHubPort()`.

```
def nextHubPort( )
```

**Returns :**

a pointer to a `YHubPort` object, corresponding to a Yocto-hub port currently online, or a `null` pointer if there are no more Yocto-hub ports to enumerate.

**hubport→registerValueCallback()**  
**hubport.registerValueCallback()****YHubPort**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

---

<b>hubport→set_enabled()</b>	<b>YHubPort</b>
<b>hubport→setEnabled()hubport.set_enabled()</b>	

---

Changes the activation of the Yocto-hub port.

```
def set_enabled( newval )
```

If the port is enabled, the connected module is powered. Otherwise, port power is shut down.

**Parameters :**

**newval** either Y\_ENABLED\_FALSE or Y\_ENABLED\_TRUE, according to the activation of the Yocto-hub port

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**hubport→set\_logicalName()**  
**hubport→setLogicalName()**  
**hubport.set\_logicalName()**

**YHubPort**

Changes the logical name of the Yocto-hub port.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the Yocto-hub port.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**hubport→set(userData)**

**YHubPort**

**hubport→setUserData()hubport.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.20. Humidity function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_humidity.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YHumidity = yoctolib.YHumidity;
php	require_once('yocto_humidity.php');
cpp	#include "yocto_humidity.h"
m	#import "yocto_humidity.h"
pas	uses yocto_humidity;
vb	yocto_humidity.vb
cs	yocto_humidity.cs
java	import com.yoctopuce.YoctoAPI.YHumidity;
py	from yocto_humidity import *

### Global functions

#### yFindHumidity(func)

Retrieves a humidity sensor for a given identifier.

#### yFirstHumidity()

Starts the enumeration of humidity sensors currently accessible.

### YHumidity methods

#### humidity→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### humidity→describe()

Returns a short text that describes unambiguously the instance of the humidity sensor in the form TYPE (NAME )=SERIAL . FUNCTIONID.

#### humidity→get\_advertisedValue()

Returns the current value of the humidity sensor (no more than 6 characters).

#### humidity→get\_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

#### humidity→get\_currentValue()

Returns the current measure for the humidity.

#### humidity→get\_errorMessage()

Returns the error message of the latest error with the humidity sensor.

#### humidity→get\_errorType()

Returns the numerical error code of the latest error with the humidity sensor.

#### humidity→get\_friendlyName()

Returns a global identifier of the humidity sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### humidity→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### humidity→get\_functionId()

Returns the hardware identifier of the humidity sensor, without reference to the module.

#### humidity→get\_hardwareId()

Returns the unique hardware identifier of the humidity sensor in the form SERIAL . FUNCTIONID.

<b>humidity→get_highestValue()</b>	Returns the maximal value observed for the humidity.
<b>humidity→get_logFrequency()</b>	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>humidity→get_logicalName()</b>	Returns the logical name of the humidity sensor.
<b>humidity→get_lowestValue()</b>	Returns the minimal value observed for the humidity.
<b>humidity→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>humidity→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>humidity→get_recordedData(startTime, endTime)</b>	Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>humidity→get_reportFrequency()</b>	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>humidity→get_resolution()</b>	Returns the resolution of the measured values.
<b>humidity→get_unit()</b>	Returns the measuring unit for the humidity.
<b>humidity→get(userData)</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>humidity→isOnline()</b>	Checks if the humidity sensor is currently reachable, without raising any error.
<b>humidity→isOnline_async(callback, context)</b>	Checks if the humidity sensor is currently reachable, without raising any error (asynchronous version).
<b>humidity→load(msValidity)</b>	Preloads the humidity sensor cache with a specified validity duration.
<b>humidity→loadCalibrationPoints(rawValues, refValues)</b>	Retrieves error correction data points previously entered using the method calibrateFromPoints.
<b>humidity→load_async(msValidity, callback, context)</b>	Preloads the humidity sensor cache with a specified validity duration (asynchronous version).
<b>humidity→nextHumidity()</b>	Continues the enumeration of humidity sensors started using yFirstHumidity().
<b>humidity→registerTimedReportCallback(callback)</b>	Registers the callback function that is invoked on every periodic timed notification.
<b>humidity→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>humidity→set_highestValue(newval)</b>	Changes the recorded maximal value observed for the humidity.
<b>humidity→set_logFrequency(newval)</b>	Changes the datalogger recording frequency for this function.
<b>humidity→set_logicalName(newval)</b>	Changes the logical name of the humidity sensor.

**humidity→set\_lowestValue(newval)**

Changes the recorded minimal value observed for the humidity.

**humidity→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**humidity→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**humidity→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**humidity→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YHumidity.FindHumidity() yFindHumidity()YHumidity.FindHumidity()

YHumidity

Retrieves a humidity sensor for a given identifier.

```
def FindHumidity( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the humidity sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YHumidity.isOnline()` to test if the humidity sensor is indeed online at a given time. In case of ambiguity when looking for a humidity sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the humidity sensor

### Returns :

a `YHumidity` object allowing you to drive the humidity sensor.

**YHumidity.FirstHumidity()****yFirstHumidity()YHumidity.FirstHumidity()****YHumidity**

Starts the enumeration of humidity sensors currently accessible.

```
def FirstHumidity( )
```

Use the method `YHumidity.nextHumidity( )` to iterate on next humidity sensors.

**Returns :**

a pointer to a `YHumidity` object, corresponding to the first humidity sensor currently online, or a null pointer if there are none.

**humidity→calibrateFromPoints()****YHumidity****humidity.calibrateFromPoints()**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**humidity→describe()humidity.describe()****YHumidity**

Returns a short text that describes unambiguously the instance of the humidity sensor in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the humidity sensor (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**humidity→get\_advertisedValue()**

**YHumidity**

**humidity→advertisedValue()**

**humidity.get\_advertisedValue()**

---

Returns the current value of the humidity sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the humidity sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**humidity→get\_currentRawValue()**  
**humidity→currentRawValue()**  
**humidity.get\_currentRawValue()**

**YHumidity**

Returns the unrounded and uncalibrated raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**humidity→get\_currentValue()**

**YHumidity**

**humidity→currentValue()humidity.get\_currentValue()**

---

Returns the current measure for the humidity.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current measure for the humidity

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

**humidity→getErrorMessage()**  
**humidity→errorMessage()**  
**humidity.getErrorMessage()****YHumidity**

---

Returns the error message of the latest error with the humidity sensor.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the humidity sensor object

### humidity→get\_errorType()

YHumidity

### humidity→errorType()humidity.get\_errorType()

---

Returns the numerical error code of the latest error with the humidity sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the humidity sensor object

**humidity→get\_friendlyName()**  
**humidity→friendlyName()**  
**humidity.get\_friendlyName()**

**YHumidity**

Returns a global identifier of the humidity sensor in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the humidity sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the humidity sensor (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the humidity sensor using logical names (ex: MyCustomName . relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

---

<b>humidity→get_functionDescriptor()</b>	<b>YHumidity</b>
<b>humidity→functionDescriptor()</b>	
<b>humidity.get_functionDescriptor()</b>	

---

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**humidity→get\_functionId()****YHumidity****humidity→functionId()humidity.get\_functionId()**

---

Returns the hardware identifier of the humidity sensor, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the humidity sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## humidity→get.hardwareId()

YHumidity

## humidity→hardwareId()humidity.get.hardwareId()

Returns the unique hardware identifier of the humidity sensor in the form SERIAL.FUNCTIONID.

```
def get.hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the humidity sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the humidity sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**humidity→get\_highestValue()**  
**humidity→highestValue()**  
**humidity.get\_highestValue()**

**YHumidity**

Returns the maximal value observed for the humidity.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the humidity

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

---

<b>humidity→get_logFrequency()</b>	<b>YHumidity</b>
<b>humidity→logFrequency()</b>	
<b>humidity.get_logFrequency()</b>	

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**humidity→get\_logicalName()****YHumidity****humidity→logicalName()humidity.get\_logicalName()**

Returns the logical name of the humidity sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the humidity sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**humidity→get\_lowestValue()**

**YHumidity**

**humidity→lowestValue()humidity.get\_lowestValue()**

---

Returns the minimal value observed for the humidity.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the humidity

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**humidity→get\_module()****YHumidity****humidity→module()humidity.get\_module()**

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

**humidity→get\_recordedData()**  
**humidity→recordedData()**  
**humidity.get\_recordedData()****YHumidity**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime )
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**humidity→get\_reportFrequency()****YHumidity****humidity→reportFrequency()****humidity.get\_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

## humidity→get\_resolution()

YHumidity

## humidity→resolution()humidity.get\_resolution()

---

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**humidity→get\_unit()****YHumidity****humidity→unit()humidity.get\_unit()**

Returns the measuring unit for the humidity.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the humidity

On failure, throws an exception or returns Y\_UNIT\_INVALID.

### humidity→get(userData)

YHumidity

### humidity→userData()humidity.get(userData)

---

Returns the value of the userData attribute, as previously stored using method set(userData).

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**humidity→isOnline()humidity.isOnline()****YHumidity**

Checks if the humidity sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the humidity sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the humidity sensor.

**Returns :**

`true` if the humidity sensor can be reached, and `false` otherwise

**humidity→load()**humidity.load()******YHumidity**

Preloads the humidity sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**humidity→loadCalibrationPoints()****YHumidity****humidity.loadCalibrationPoints()**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## humidity→nextHumidity()**humidity.nextHumidity()**

**YHumidity**

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

```
def nextHumidity( )
```

**Returns :**

a pointer to a `YHumidity` object, corresponding to a humidity sensor currently online, or a null pointer if there are no more humidity sensors to enumerate.

**humidity→registerTimedReportCallback()**  
**humidity.registerTimedReportCallback()****YHumidity**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**humidity→registerValueCallback()**  
**humidity.registerValueCallback()****YHumidity**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**humidity→set\_highestValue()**  
**humidity→setHighestValue()**  
**humidity.set\_highestValue()**

**YHumidity**

Changes the recorded maximal value observed for the humidity.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed for the humidity

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**humidity→set\_logFrequency()**  
**humidity→setLogFrequency()**  
**humidity.set\_logFrequency()**

YHumidity

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**humidity→set\_logicalName()**  
**humidity→setLogicalName()**  
**humidity.set\_logicalName()**

**YHumidity**

Changes the logical name of the humidity sensor.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the humidity sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**humidity→set\_lowestValue()**  
**humidity→setLowestValue()**  
**humidity.set\_lowestValue()**

YHumidity

---

Changes the recorded minimal value observed for the humidity.

```
def set_lowestValue( newval )
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed for the humidity

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**humidity→set\_reportFrequency()**  
**humidity→setReportFrequency()**  
**humidity.set\_reportFrequency()**

**YHumidity**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**humidity→set\_resolution()****YHumidity****humidity→setResolution()humidity.set\_resolution()**

Changes the resolution of the measured physical values.

```
def set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**humidity→set(userData)****YHumidity****humidity→setUserData()humidity.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData) data
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.21. Led function interface

Yoctopuce application programming interface allows you not only to drive the intensity of the led, but also to have it blink at various preset frequencies.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_led.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YLed = yoctolib.YLed;
php	require_once('yocto_led.php');
cpp	#include "yocto_led.h"
m	#import "yocto_led.h"
pas	uses yocto_led;
vb	yocto_led.vb
cs	yocto_led.cs
java	import com.yoctopuce.YoctoAPI.YLed;
py	from yocto_led import *

### Global functions

#### yFindLed(func)

Retrieves a led for a given identifier.

#### yFirstLed()

Starts the enumeration of leds currently accessible.

### YLed methods

#### led->describe()

Returns a short text that describes unambiguously the instance of the led in the form TYPE (NAME )=SERIAL .FUNCTIONID.

#### led->get\_advertisedValue()

Returns the current value of the led (no more than 6 characters).

#### led->get\_blinking()

Returns the current led signaling mode.

#### led->get\_errorMessage()

Returns the error message of the latest error with the led.

#### led->get\_errorType()

Returns the numerical error code of the latest error with the led.

#### led->get\_friendlyName()

Returns a global identifier of the led in the format MODULE \_ NAME . FUNCTION \_ NAME.

#### led->get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### led->get\_functionId()

Returns the hardware identifier of the led, without reference to the module.

#### led->get\_hardwareId()

Returns the unique hardware identifier of the led in the form SERIAL .FUNCTIONID.

#### led->get\_logicalName()

Returns the logical name of the led.

#### led->get\_luminosity()

Returns the current led intensity (in per cent).

#### led->get\_module()

Gets the YModule object for the device on which the function is located.

**led->get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**led->get\_power()**

Returns the current led state.

**led->get\_userData()**

Returns the value of the userData attribute, as previously stored using method set(userData).

**led->isOnline()**

Checks if the led is currently reachable, without raising any error.

**led->isOnline\_async(callback, context)**

Checks if the led is currently reachable, without raising any error (asynchronous version).

**led->load(msValidity)**

Preloads the led cache with a specified validity duration.

**led->load\_async(msValidity, callback, context)**

Preloads the led cache with a specified validity duration (asynchronous version).

**led->nextLed()**

Continues the enumeration of leds started using yFirstLed( ).

**led->registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**led->set\_blinking(newval)**

Changes the current led signaling mode.

**led->set\_logicalName(newval)**

Changes the logical name of the led.

**led->set\_luminosity(newval)**

Changes the current led intensity (in per cent).

**led->set\_power(newval)**

Changes the state of the led.

**led->set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**led->wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YLed.FindLed() yFindLed()YLed.FindLed()

YLed

Retrieves a led for a given identifier.

```
def FindLed( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the led is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLed.isOnline()` to test if the led is indeed online at a given time. In case of ambiguity when looking for a led by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the led

### Returns :

a `YLed` object allowing you to drive the led.

## YLed.FirstLed() yFirstLed()YLed.FirstLed()

YLed

Starts the enumeration of leds currently accessible.

```
def FirstLed( )
```

Use the method `YLed.nextLed()` to iterate on next leds.

**Returns :**

a pointer to a `YLed` object, corresponding to the first led currently online, or a `null` pointer if there are none.

**led→describe()led.describe()**

YLed

Returns a short text that describes unambiguously the instance of the led in the form TYPE (NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the led (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

---

**led→get\_advertisedValue()****YLed****led→advertisedValue()led.get\_advertisedValue()**

Returns the current value of the led (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the led (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**led→get\_blinking()  
led→blinking()led.get\_blinking()**

YLed

Returns the current led signaling mode.

```
def get_blinking( )
```

**Returns :**

a value among Y\_BLINKING\_STILL, Y\_BLINKING\_RELAX, Y\_BLINKING\_AWARE, Y\_BLINKING\_RUN, Y\_BLINKING\_CALL and Y\_BLINKING\_PANIC corresponding to the current led signaling mode

On failure, throws an exception or returns Y\_BLINKING\_INVALID.

---

**led->getErrorMessage()****YLed****led->errorMessage()led.getErrorMessage()**

---

Returns the error message of the latest error with the led.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the led object

**led->get\_errorType()**

YLed

**led->errorType()led.get\_errorType()**

---

Returns the numerical error code of the latest error with the led.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the led object

**led->get\_friendlyName()****YLed****led->friendlyName()led.get\_friendlyName()**

Returns a global identifier of the led in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the led if they are defined, otherwise the serial number of the module and the hardware identifier of the led (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the led using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**led→get\_functionDescriptor()**  
**led→functionDescriptor()**  
**led.get\_functionDescriptor()**

YLed

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

---

**led->get\_functionId()****YLed****led->functionId()led.get\_functionId()**

---

Returns the hardware identifier of the led, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the led (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**led→get\_hardwareId()**

YLed

**led→hardwareId()led.get\_hardwareId()**

---

Returns the unique hardware identifier of the led in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the led. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the led (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**led→get\_logicalName()****YLed****led→logicalName()led.get\_logicalName()**

Returns the logical name of the led.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the led. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

## **led→get\_luminosity()**

YLed

## **led→luminosity()led.get\_luminosity()**

---

Returns the current led intensity (in per cent).

```
def get_luminosity( )
```

**Returns :**

an integer corresponding to the current led intensity (in per cent)

On failure, throws an exception or returns Y\_LUMINOSITY\_INVALID.

**led→get\_module()****YLed****led→module()led.get\_module()**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**led→get\_power()**

YLed

**led→power()led.get\_power()**

---

Returns the current led state.

```
def get_power( )
```

**Returns :**

either Y\_POWER\_OFF or Y\_POWER\_ON, according to the current led state

On failure, throws an exception or returns Y\_POWER\_INVALID.

---

**led→get(userData)****YLed****led→userData()led.get(userData())**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**led→isOnline()led.isOnline()****YLed**

Checks if the led is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the led in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the led.

**Returns :**

`true` if the led can be reached, and `false` otherwise

**led→load()|led.load()****YLed**

Preloads the led cache with a specified validity duration.

```
def load( msValidity )
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## led->nextLed()|led.nextLed()

YLed

Continues the enumeration of leds started using `yFirstLed()`.

```
def nextLed( )
```

**Returns :**

a pointer to a YLed object, corresponding to a led currently online, or a `null` pointer if there are no more leds to enumerate.

**led→registerValueCallback()  
led.registerValueCallback()****YLed**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**led→set\_blinking()** YLed  
**led→setBlinking()led.set\_blinking()**

---

Changes the current led signaling mode.

```
def set_blinking( newval )
```

**Parameters :**

**newval** a value among Y\_BLINKING\_STILL, Y\_BLINKING\_RELAX, Y\_BLINKING\_AWARE, Y\_BLINKING\_RUN, Y\_BLINKING\_CALL and Y\_BLINKING\_PANIC corresponding to the current led signaling mode

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**led->set\_logicalName()**

**YLed**

**led->setLogicalName()led.set\_logicalName()**

Changes the logical name of the led.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the led.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**led->set\_luminosity()**

YLed

**led->setLuminosity()led.set\_luminosity()**

---

Changes the current led intensity (in per cent).

```
def set_luminosity( newval )
```

**Parameters :**

**newval** an integer corresponding to the current led intensity (in per cent)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**led->set\_power()****YLed****led->setPower()|led.set\_power()**

Changes the state of the led.

```
def set_power( newval )
```

**Parameters :**

**newval** either Y\_POWER\_OFF or Y\_POWER\_ON, according to the state of the led

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**led→set(userData)**  
**led→setUserData()|led.set(userData)**

---

YLed

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData|setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.22. LightSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_lightsensor.js'></script>
node.js	var yoctolib = require('yoctolib');
php	var YLightSensor = yoctolib.YLightSensor;
cpp	require_once('yocto_lightsensor.php');
m	#include "yocto_lightsensor.h"
pas	#import "yocto_lightsensor.h"
vb	uses yocto_lightsensor;
cs	yocto_lightsensor.vb
java	yocto_lightsensor.cs
py	import com.yoctopuce.YoctoAPI.YLightSensor;
	from yocto_lightsensor import *

### Global functions

#### yFindLightSensor(func)

Retrieves a light sensor for a given identifier.

#### yFirstLightSensor()

Starts the enumeration of light sensors currently accessible.

### YLightSensor methods

#### lightsensor→calibrate(calibratedVal)

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

#### lightsensor→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### lightsensor→describe()

Returns a short text that describes unambiguously the instance of the light sensor in the form TYPE (NAME) = SERIAL.FUNCTIONID.

#### lightsensor→get\_advertisedValue()

Returns the current value of the light sensor (no more than 6 characters).

#### lightsensor→get\_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

#### lightsensor→get\_currentValue()

Returns the current measure for the ambient light.

#### lightsensor→get\_errorMessage()

Returns the error message of the latest error with the light sensor.

#### lightsensor→get\_errorType()

Returns the numerical error code of the latest error with the light sensor.

#### lightsensor→get\_friendlyName()

Returns a global identifier of the light sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### lightsensor→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### lightsensor→get\_functionId()

Returns the hardware identifier of the light sensor, without reference to the module.
<b>lightsensor→get_hardwareId()</b>
Returns the unique hardware identifier of the light sensor in the form SERIAL.FUNCTIONID.
<b>lightsensor→get_highestValue()</b>
Returns the maximal value observed for the ambient light.
<b>lightsensor→get_logFrequency()</b>
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>lightsensor→get_logicalName()</b>
Returns the logical name of the light sensor.
<b>lightsensor→get_lowestValue()</b>
Returns the minimal value observed for the ambient light.
<b>lightsensor→get_module()</b>
Gets the YModule object for the device on which the function is located.
<b>lightsensor→get_module_async(callback, context)</b>
Gets the YModule object for the device on which the function is located (asynchronous version).
<b>lightsensor→get_recordedData(startTime, endTime)</b>
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>lightsensor→get_reportFrequency()</b>
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>lightsensor→get_resolution()</b>
Returns the resolution of the measured values.
<b>lightsensor→get_unit()</b>
Returns the measuring unit for the ambient light.
<b>lightsensor→get(userData)</b>
Returns the value of the userData attribute, as previously stored using method set(userData).
<b>lightsensor→isOnline()</b>
Checks if the light sensor is currently reachable, without raising any error.
<b>lightsensor→isOnline_async(callback, context)</b>
Checks if the light sensor is currently reachable, without raising any error (asynchronous version).
<b>lightsensor→load(msValidity)</b>
Preloads the light sensor cache with a specified validity duration.
<b>lightsensor→loadCalibrationPoints(rawValues, refValues)</b>
Retrieves error correction data points previously entered using the method calibrateFromPoints.
<b>lightsensor→load_async(msValidity, callback, context)</b>
Preloads the light sensor cache with a specified validity duration (asynchronous version).
<b>lightsensor→nextLightSensor()</b>
Continues the enumeration of light sensors started using yFirstLightSensor( ).
<b>lightsensor→registerTimedReportCallback(callback)</b>
Registers the callback function that is invoked on every periodic timed notification.
<b>lightsensor→registerValueCallback(callback)</b>
Registers the callback function that is invoked on every change of advertised value.
<b>lightsensor→set_highestValue(newval)</b>
Changes the recorded maximal value observed for the ambient light.
<b>lightsensor→set_logFrequency(newval)</b>

Changes the datalogger recording frequency for this function.

**lightsensor→set\_logicalName(newval)**

Changes the logical name of the light sensor.

**lightsensor→set\_lowestValue(newval)**

Changes the recorded minimal value observed for the ambient light.

**lightsensor→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**lightsensor→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**lightsensor→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**lightsensor→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YLightSensor.FindLightSensor() yFindLightSensor()YLightSensor.FindLightSensor()

YLightSensor

Retrieves a light sensor for a given identifier.

```
def FindLightSensor( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the light sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YLightSensor.isOnline()` to test if the light sensor is indeed online at a given time. In case of ambiguity when looking for a light sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the light sensor

### Returns :

a `YLightSensor` object allowing you to drive the light sensor.

**YLightSensor.FirstLightSensor()****YLightSensor****yFirstLightSensor()YLightSensor.FirstLightSensor()**

Starts the enumeration of light sensors currently accessible.

```
def FirstLightSensor( )
```

Use the method `YLightSensor.nextLightSensor( )` to iterate on next light sensors.

**Returns :**

a pointer to a `YLightSensor` object, corresponding to the first light sensor currently online, or a null pointer if there are none.

**lightsensor→calibrate()lightsensor.calibrate()****YLightSensor**

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

```
def calibrate( calibratedVal)
```

**Parameters :**

**calibratedVal** the desired target value.

**Returns :**

**YAPI\_SUCCESS** if the call succeeds.

On failure, throws an exception or returns a negative error code.

**lightsensor→calibrateFromPoints()**  
**lightsensor.calibrateFromPoints()****YLightSensor**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**lightsensor→describe()lightsensor.describe()****YLightSensor**

Returns a short text that describes unambiguously the instance of the light sensor in the form  
TYPE ( NAME )=SERIAL . FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the light sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

**lightsensor→get\_advertisedValue()**  
**lightsensor→advertisedValue()**  
**lightsensor.get\_advertisedValue()**

**YLightSensor**

Returns the current value of the light sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the light sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**lightsensor→get\_currentRawValue()**  
**lightsensor→currentRawValue()**  
**lightsensor.get\_currentRawValue()**

**YLightSensor**

Returns the unrounded and uncalibrated raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**lightsensor→get\_currentValue()**  
**lightsensor→currentValue()**  
**lightsensor.get\_currentValue()**

**YLightSensor**

Returns the current measure for the ambient light.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current measure for the ambient light

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

**lightsensor→get\_errorMessage()**  
**lightsensor→errorMessage()**  
**lightsensor.get\_errorMessage()**

**YLightSensor**

Returns the error message of the latest error with the light sensor.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the light sensor object

---

**lightsensor→get\_errorType()****YLightSensor****lightsensor→errorType()lightsensor.get\_errorType()**

---

Returns the numerical error code of the latest error with the light sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the light sensor object

**lightsensor→get\_friendlyName()**  
**lightsensor→friendlyName()**  
**lightsensor.get\_friendlyName()**

**YLightSensor**

Returns a global identifier of the light sensor in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the light sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the light sensor (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the light sensor using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**lightsensor→get\_functionDescriptor()****YLightSensor****lightsensor→functionDescriptor()****lightsensor.get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**lightsensor→get\_functionId()**

**YLightSensor**

**lightsensor→functionId()lightsensor.get\_functionId()**

---

Returns the hardware identifier of the light sensor, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the light sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**lightsensor→get\_hwId()**  
**lightsensor→hardwareId()**  
**lightsensor.get\_hwId()**

**YLightSensor**

Returns the unique hardware identifier of the light sensor in the form SERIAL.FUNCTIONID.

**def get\_hwId( )**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the light sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the light sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**lightsensor→get\_highestValue()**  
**lightsensor→highestValue()**  
**lightsensor.get\_highestValue()**

**YLightSensor**

Returns the maximal value observed for the ambient light.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the ambient light

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**lightsensor→get\_logFrequency()**  
**lightsensor→logFrequency()**  
**lightsensor.get\_logFrequency()**

**YLightSensor**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**lightsensor→get\_logicalName()**  
**lightsensor→logicalName()**  
**lightsensor.get\_logicalName()**

**YLightSensor**

Returns the logical name of the light sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the light sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**lightsensor→get\_lowestValue()**  
**lightsensor→lowestValue()**  
**lightsensor.get\_lowestValue()**

**YLightSensor**

Returns the minimal value observed for the ambiant light.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the ambiant light

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**lightsensor→get\_module()**

**YLightSensor**

**lightsensor→module()lightsensor.get\_module()**

---

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**lightsensor→get\_recordedData()**  
**lightsensor→recordedData()**  
**lightsensor.get\_recordedData()**

**YLightSensor**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**lightsensor→get\_reportFrequency()**  
**lightsensor→reportFrequency()**  
**lightsensor.get\_reportFrequency()**

**YLightSensor**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

---

**lightsensor→get\_resolution()****YLightSensor****lightsensor→resolution()lightsensor.get\_resolution()**

---

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**lightsensor→get\_unit()**

**YLightSensor**

**lightsensor→unit()lightsensor.get\_unit()**

---

Returns the measuring unit for the ambient light.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the ambient light

On failure, throws an exception or returns Y\_UNIT\_INVALID.

---

**lightsensor→get(userData)****YLightSensor****lightsensor→userData()lightsensor.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**lightsensor→isOnline()****lightsensor.isOnline()****YLightSensor**

Checks if the light sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the light sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the light sensor.

**Returns :**

true if the light sensor can be reached, and false otherwise

**lightsensor→load()lightsensor.load()****YLightSensor**

Preloads the light sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**lightsensor→loadCalibrationPoints()**  
**lightsensor.loadCalibrationPoints()****YLightSensor**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**lightsensor→nextLightSensor()**  
**lightsensor.nextLightSensor()****YLightSensor**

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

```
def nextLightSensor( )
```

**Returns :**

a pointer to a `YLightSensor` object, corresponding to a light sensor currently online, or a `null` pointer if there are no more light sensors to enumerate.

**lightsensor→registerTimedReportCallback()**  
**lightsensor.registerTimedReportCallback()****YLightSensor**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**lightsensor→registerValueCallback()  
lightsensor.registerValueCallback()****YLightSensor**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**lightsensor→set\_highestValue()**  
**lightsensor→setHighestValue()**  
**lightsensor.set\_highestValue()**

**YLightSensor**

Changes the recorded maximal value observed for the ambiant light.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed for the ambiant light

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**lightsensor→set\_logFrequency()**  
**lightsensor→setLogFrequency()**  
**lightsensor.set\_logFrequency()**

**YLightSensor**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**lightsensor→set\_logicalName()**  
**lightsensor→setLogicalName()**  
**lightsensor.set\_logicalName()**

**YLightSensor**

Changes the logical name of the light sensor.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the light sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**lightsensor→set\_lowestValue()**  
**lightsensor→setLowestValue()**  
**lightsensor.set\_lowestValue()**

**YLightSensor**

Changes the recorded minimal value observed for the ambient light.

```
def set_lowestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed for the ambient light

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**lightsensor→set\_reportFrequency()**  
**lightsensor→setReportFrequency()**  
**lightsensor.set\_reportFrequency()**

**YLightSensor**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**lightsensor→set\_resolution()**  
**lightsensor→setResolution()**  
**lightsensor.set\_resolution()**

**YLightSensor**

Changes the resolution of the measured physical values.

```
def set_resolution( newval )
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**lightsensor→set(userData)**  
**lightsensor→setUserData()**  
**lightsensor.set(userData)**

**YLightSensor**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData) data
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.23. Magnetometer function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_magnetometer.js'></script>
nodejs	var yoctolib = require('yoctolib');
php	var YMagnetometer = yoctolib.YMagnetometer;
cpp	require_once('yocto_magnetometer.php');
m	#include "yocto_magnetometer.h"
pas	#import "yocto_magnetometer.h"
vb	uses yocto_magnetometer;
cs	yocto_magnetometer.vb
java	yocto_magnetometer.cs
py	import com.yoctopuce.YoctoAPI.YMagnetometer;
	from yocto_magnetometer import *

### Global functions

#### **yFindMagnetometer(func)**

Retrieves a magnetometer for a given identifier.

#### **yFirstMagnetometer()**

Starts the enumeration of magnetometers currently accessible.

### YMagnetometer methods

#### **magnetometer→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### **magnetometer→describe()**

Returns a short text that describes unambiguously the instance of the magnetometer in the form TYPE (NAME) = SERIAL . FUNCTIONID.

#### **magnetometer→get\_advertisedValue()**

Returns the current value of the magnetometer (no more than 6 characters).

#### **magnetometer→get\_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### **magnetometer→get\_currentValue()**

Returns the current value of the magnetic field.

#### **magnetometer→get\_errorMessage()**

Returns the error message of the latest error with the magnetometer.

#### **magnetometer→get\_errorType()**

Returns the numerical error code of the latest error with the magnetometer.

#### **magnetometer→get\_friendlyName()**

Returns a global identifier of the magnetometer in the format MODULE\_NAME . FUNCTION\_NAME.

#### **magnetometer→get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### **magnetometer→get\_functionId()**

Returns the hardware identifier of the magnetometer, without reference to the module.

#### **magnetometer→get\_hardwareId()**

Returns the unique hardware identifier of the magnetometer in the form SERIAL . FUNCTIONID.

<b>magnetometer→get_highestValue()</b>	Returns the maximal value observed for the magnetic field since the device was started.
<b>magnetometer→get_logFrequency()</b>	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>magnetometer→get_logicalName()</b>	Returns the logical name of the magnetometer.
<b>magnetometer→get_lowestValue()</b>	Returns the minimal value observed for the magnetic field since the device was started.
<b>magnetometer→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>magnetometer→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>magnetometer→get_recordedData(startTime, endTime)</b>	Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>magnetometer→get_reportFrequency()</b>	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>magnetometer→get_resolution()</b>	Returns the resolution of the measured values.
<b>magnetometer→get_unit()</b>	Returns the measuring unit for the magnetic field.
<b>magnetometer→get_userData()</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>magnetometer→get_xValue()</b>	Returns the X component of the magnetic field, as a floating point number.
<b>magnetometer→get_yValue()</b>	Returns the Y component of the magnetic field, as a floating point number.
<b>magnetometer→get_zValue()</b>	Returns the Z component of the magnetic field, as a floating point number.
<b>magnetometer→isOnline()</b>	Checks if the magnetometer is currently reachable, without raising any error.
<b>magnetometer→isOnline_async(callback, context)</b>	Checks if the magnetometer is currently reachable, without raising any error (asynchronous version).
<b>magnetometer→load(msValidity)</b>	Preloads the magnetometer cache with a specified validity duration.
<b>magnetometer→loadCalibrationPoints(rawValues, refValues)</b>	Retrieves error correction data points previously entered using the method calibrateFromPoints.
<b>magnetometer→load_async(msValidity, callback, context)</b>	Preloads the magnetometer cache with a specified validity duration (asynchronous version).
<b>magnetometer→nextMagnetometer()</b>	Continues the enumeration of magnetometers started using yFirstMagnetometer( ).
<b>magnetometer→registerTimedReportCallback(callback)</b>	Registers the callback function that is invoked on every periodic timed notification.
<b>magnetometer→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.

**magnetometer→set\_highestValue(newval)**

Changes the recorded maximal value observed.

**magnetometer→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**magnetometer→set\_logicalName(newval)**

Changes the logical name of the magnetometer.

**magnetometer→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**magnetometer→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**magnetometer→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**magnetometer→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**magnetometer→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## Y Magnetometer.FindMagnetometer() yFindMagnetometer() Y Magnetometer.FindMagnetometer()

**Y Magnetometer**

Retrieves a magnetometer for a given identifier.

```
def FindMagnetometer( func )
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the magnetometer is online at the time it is invoked. The returned object is nevertheless valid. Use the method `Y Magnetometer.isOnline()` to test if the magnetometer is indeed online at a given time. In case of ambiguity when looking for a magnetometer by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the magnetometer

**Returns :**

a `Y Magnetometer` object allowing you to drive the magnetometer.

**Y Magnetometer.FirstMagnetometer()  
yFirstMagnetometer()  
Y Magnetometer.FirstMagnetometer()****Y Magnetometer**

Starts the enumeration of magnetometers currently accessible.

```
def FirstMagnetometer( )
```

Use the method `Y Magnetometer.nextMagnetometer()` to iterate on next magnetometers.

**Returns :**

a pointer to a `Y Magnetometer` object, corresponding to the first magnetometer currently online, or a null pointer if there are none.

**magnetometer→calibrateFromPoints()**  
**magnetometer.calibrateFromPoints()****YMagnetometer**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**magnetometer→describe()magnetometer.describe()****YMagnetometer**

Returns a short text that describes unambiguously the instance of the magnetometer in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the magnetometer (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**magnetometer→get\_advertisedValue()**  
**magnetometer→advertisedValue()**  
**magnetometer.get\_advertisedValue()**

---

**YMagnetometer**

Returns the current value of the magnetometer (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the magnetometer (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**magnetometer→get\_currentRawValue()**  
**magnetometer→currentRawValue()**  
**magnetometer.get\_currentRawValue()**

**YMagnetometer**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**magnetometer→get\_currentValue()**  
**magnetometer→currentValue()**  
**magnetometer.get\_currentValue()**

---

**YMagnetometer**

Returns the current value of the magnetic field.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the magnetic field

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

---

**magnetometer→get\_errorMessage()**  
**magnetometer→errorMessage()**  
**magnetometer.get\_errorMessage()**

**YMagnetometer**

Returns the error message of the latest error with the magnetometer.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the magnetometer object

**magnetometer→get\_errorType()**  
**magnetometer→errorType()**  
**magnetometer.get\_errorType()**

**YMagnetometer**

---

Returns the numerical error code of the latest error with the magnetometer.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the magnetometer object

**magnetometer→get\_friendlyName()**  
**magnetometer→friendlyName()**  
**magnetometer.get\_friendlyName()**

**YMagnetometer**

Returns a global identifier of the magnetometer in the format MODULE\_NAME . FUNCTION\_NAME.

**def get\_friendlyName( )**

The returned string uses the logical names of the module and of the magnetometer if they are defined, otherwise the serial number of the module and the hardware identifier of the magnetometer (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the magnetometer using logical names (ex: MyCustomName . relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**magnetometer→get\_functionDescriptor()**  
**magnetometer→functionDescriptor()**  
**magnetometer.get\_functionDescriptor()**

**YMagnetometer**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**magnetometer→get\_functionId()**  
**magnetometer→functionId()**  
**magnetometer.get\_functionId()**

**YMagnetometer**

Returns the hardware identifier of the magnetometer, without reference to the module.

```
def get_functionId( )
```

For example relay1

**Returns :**

a string that identifies the magnetometer (ex: relay1) On failure, throws an exception or returns Y\_FUNCTIONID\_INVALID.

**magnetometer→get\_hardwareId()**  
**magnetometer→hardwareId()**  
**magnetometer.get\_hardwareId()**

**YMagnetometer**

Returns the unique hardware identifier of the magnetometer in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the magnetometer. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the magnetometer (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**magnetometer→get\_highestValue()**  
**magnetometer→highestValue()**  
**magnetometer.get\_highestValue()**

**YMagnetometer**

Returns the maximal value observed for the magnetic field since the device was started.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the magnetic field since the device was started

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**magnetometer→get\_logFrequency()**  
**magnetometer→logFrequency()**  
**magnetometer.get\_logFrequency()**

**YMagnetometer**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**magnetometer→get\_logicalName()**  
**magnetometer→logicalName()**  
**magnetometer.get\_logicalName()**

**YMagnetometer**

Returns the logical name of the magnetometer.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the magnetometer. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**magnetometer→get\_lowestValue()**  
**magnetometer→lowestValue()**  
**magnetometer.get\_lowestValue()**

**YMagnetometer**

---

Returns the minimal value observed for the magnetic field since the device was started.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the magnetic field since the device was started

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**magnetometer→get\_module()**  
**magnetometer→module()**  
**magnetometer.get\_module()**

**YMagnetometer**

Gets the **YModule** object for the device on which the function is located.

**def get\_module( )**

If the function cannot be located on any module, the returned instance of **YModule** is not shown as online.

**Returns :**

an instance of **YModule**

**magnetometer→get\_recordedData()**  
**magnetometer→recordedData()**  
**magnetometer.get\_recordedData()**

**YMagnetometer**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime )
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**magnetometer→get\_reportFrequency()**  
**magnetometer→reportFrequency()**  
**magnetometer.get\_reportFrequency()**

**YMagnetometer**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**magnetometer→get\_resolution()**  
**magnetometer→resolution()**  
**magnetometer.get\_resolution()**

**YMagnetometer**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

---

**magnetometer→get\_unit()****YMagnetometer****magnetometer→unit()magnetometer.get\_unit()**

---

Returns the measuring unit for the magnetic field.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the magnetic field

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**magnetometer→get(userData)**  
**magnetometer→userData()**  
**magnetometer.get(userData)**

---

**YMagnetometer**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

---

**magnetometer→get\_xValue()****YMagnetometer****magnetometer→xValue()magnetometer.get\_xValue()**

---

Returns the X component of the magnetic field, as a floating point number.

```
def get_xValue( )
```

**Returns :**

a floating point number corresponding to the X component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y\_XVALUE\_INVALID.

**magnetometer→get\_yValue()**

**YMagnetometer**

**magnetometer→yValue()magnetometer.get\_yValue()**

---

Returns the Y component of the magnetic field, as a floating point number.

```
def get_yValue( )
```

**Returns :**

a floating point number corresponding to the Y component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y\_YVALUE\_INVALID.

**magnetometer→get\_zValue()****YMagnetometer****magnetometer→zValue()magnetometer.get\_zValue()**

Returns the Z component of the magnetic field, as a floating point number.

```
def get_zValue( )
```

**Returns :**

a floating point number corresponding to the Z component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y\_ZVALUE\_INVALID.

**magnetometer→isOnline()magnetometer.isOnline()****YMagnetometer**

Checks if the magnetometer is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the magnetometer in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the magnetometer.

**Returns :**

true if the magnetometer can be reached, and false otherwise

**magnetometer→load()magnetometer.load()****YMagnetometer**

Preloads the magnetometer cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**magnetometer→loadCalibrationPoints()**  
**magnetometer.loadCalibrationPoints()****YMagnetometer**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**magnetometer→nextMagnetometer()**  
**magnetometer.nextMagnetometer()****Y Magnetometer**

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

```
def nextMagnetometer( )
```

**Returns :**

a pointer to a `Y Magnetometer` object, corresponding to a magnetometer currently online, or a null pointer if there are no more magnetometers to enumerate.

**magnetometer→registerTimedReportCallback()**  
**magnetometer.registerTimedReportCallback()****YMagnetometer**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**magnetometer→registerValueCallback()**  
**magnetometer.registerValueCallback()****YMagnetometer**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**magnetometer→set\_highestValue()**  
**magnetometer→setHighestValue()**  
**magnetometer.set\_highestValue()**

**YMagnetometer**

---

Changes the recorded maximal value observed.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**magnetometer→set\_logFrequency()**  
**magnetometer→setLogFrequency()**  
**magnetometer.set\_logFrequency()**

**YMagmeter**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**magnetometer→set\_logicalName()**  
**magnetometer→setLogicalName()**  
**magnetometer.set\_logicalName()**

**YMagnetometer**

Changes the logical name of the magnetometer.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the magnetometer.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**magnetometer→set\_lowestValue()**  
**magnetometer→setLowestValue()**  
**magnetometer.set\_lowestValue()**

**YMagnetometer**

Changes the recorded minimal value observed.

```
def set_lowestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**magnetometer→set\_reportFrequency()**  
**magnetometer→setReportFrequency()**  
**magnetometer.set\_reportFrequency()****YMagnetometer**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**magnetometer→set\_resolution()**  
**magnetometer→setResolution()**  
**magnetometer.set\_resolution()**

**YMagnetometer**

Changes the resolution of the measured physical values.

```
def set_resolution( newval )
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**magnetometer→set(userData)**  
**magnetometer→setUserData()**  
**magnetometer.set(userData)**

---

**YMagnetometer**

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.24. Measured value

YMeasure objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the YDataSet class.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
node.js	var yoctolib = require('yoctolib');
	var YAPI = yoctolib.YAPI;
	var YModule = yoctolib.YModule;
php	require_once('yocto_api.php');
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

### YMeasure methods

#### **measure→get\_averageValue()**

Returns the average value observed during the time interval covered by this measure.

#### **measure→get\_endTimeUTC()**

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

#### **measure→get\_maxValue()**

Returns the largest value observed during the time interval covered by this measure.

#### **measure→get\_minValue()**

Returns the smallest value observed during the time interval covered by this measure.

#### **measure→get\_startTimeUTC()**

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

**measure→get\_averageValue()**  
**measure→averageValue()**  
**measure.get\_averageValue()**

---

**YMeasure**

Returns the average value observed during the time interval covered by this measure.

```
def get_averageValue( )
```

**Returns :**

a floating-point number corresponding to the average value observed.

---

**measure→get\_endTimeUTC()****YMeasure****measure→endTimeUTC()measure.get\_endTimeUTC()**

---

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

```
def get_endTimeUTC( )
```

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

**Returns :**

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.

**measure→get\_maxValue()**

**YMeasure**

**measure→maxValue()measure.get\_maxValue()**

---

Returns the largest value observed during the time interval covered by this measure.

```
def get_maxValue( )
```

**Returns :**

a floating-point number corresponding to the largest value observed.

---

**measure→get\_minValue()****YMeasure****measure→minValue()measure.get\_minValue()**

---

Returns the smallest value observed during the time interval covered by this measure.

```
def get_minValue( )
```

**Returns :**

a floating-point number corresponding to the smallest value observed.

**measure→getStartTimeUTC()**  
**measure→startTimeUTC()**  
**measure.getStartTimeUTC()**

---

**YMeasure**

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

```
def getStartTimeUTC( )
```

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

**Returns :**

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

## 3.25. Module control interface

This interface is identical for all Yoctopuce USB modules. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>
node.js	var yoctolib = require('yoctolib');
	var YAPI = yoctolib.YAPI;
	var YModule = yoctolib.YModule;
php	require_once('yocto_api.php');
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

### Global functions

#### yFindModule(func)

Allows you to find a module from its serial number or from its logical name.

#### yFirstModule()

Starts the enumeration of modules currently accessible.

### YModule methods

#### module→describe()

Returns a descriptive text that identifies the module.

#### module→download(pathname)

Downloads the specified built-in file and returns a binary buffer with its content.

#### module→functionCount()

Returns the number of functions (beside the "module" interface) available on the module.

#### module→functionId(functionIndex)

Retrieves the hardware identifier of the *n*th function on the module.

#### module→functionName(functionIndex)

Retrieves the logical name of the *n*th function on the module.

#### module→functionValue(functionIndex)

Retrieves the advertised value of the *n*th function on the module.

#### module→get\_beacon()

Returns the state of the localization beacon.

#### module→get\_errorMessage()

Returns the error message of the latest error with this module object.

#### module→get\_errorType()

Returns the numerical error code of the latest error with this module object.

#### module→get\_firmwareRelease()

Returns the version of the firmware embedded in the module.

#### module→get\_hardwareId()

Returns the unique hardware identifier of the module.

#### module→get\_icon2d()

### 3. Reference

Returns the icon of the module.
<b>module→get_lastLogs()</b> Returns a string with last logs of the module.
<b>module→get_logicalName()</b> Returns the logical name of the module.
<b>module→get_luminosity()</b> Returns the luminosity of the module informative leds (from 0 to 100).
<b>module→get_persistentSettings()</b> Returns the current state of persistent module settings.
<b>module→get_productId()</b> Returns the USB device identifier of the module.
<b>module→get_productName()</b> Returns the commercial name of the module, as set by the factory.
<b>module→get_productRelease()</b> Returns the hardware release version of the module.
<b>module→get_rebootCountdown()</b> Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.
<b>module→get_serialNumber()</b> Returns the serial number of the module, as set by the factory.
<b>module→get_upTime()</b> Returns the number of milliseconds spent since the module was powered on.
<b>module→get_usbBandwidth()</b> Returns the number of USB interfaces used by the module.
<b>module→get_usbCurrent()</b> Returns the current consumed by the module on the USB bus, in milli-amps.
<b>module→get(userData)</b> Returns the value of the userData attribute, as previously stored using method <code>set(userData)</code> .
<b>module→isOnline()</b> Checks if the module is currently reachable, without raising any error.
<b>module→isOnline_async(callback, context)</b> Checks if the module is currently reachable, without raising any error.
<b>module→load(msValidity)</b> Preloads the module cache with a specified validity duration.
<b>module→load_async(msValidity, callback, context)</b> Preloads the module cache with a specified validity duration (asynchronous version).
<b>module→nextModule()</b> Continues the module enumeration started using <code>yFirstModule()</code> .
<b>module→reboot(secBeforeReboot)</b> Schedules a simple module reboot after the given number of seconds.
<b>module→registerLogCallback(callback)</b> todo
<b>module→revertFromFlash()</b> Reloads the settings stored in the nonvolatile memory, as when the module is powered on.
<b>module→saveToFlash()</b> Saves current settings in the nonvolatile memory of the module.

**module→set\_beacon(newval)**

Turns on or off the module localization beacon.

**module→set\_logicalName(newval)**

Changes the logical name of the module.

**module→set\_luminosity(newval)**

Changes the luminosity of the module informative leds.

**module→set\_usbBandwidth(newval)**

Changes the number of USB interfaces used by the module.

**module→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**module→triggerFirmwareUpdate(secBeforeReboot)**

Schedules a module reboot into special firmware update mode.

**module→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YModule.FindModule() yFindModule()YModule.FindModule()

**YModule**

Allows you to find a module from its serial number or from its logical name.

```
def FindModule( func)
```

This function does not require that the module is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YModule.isOnline()` to test if the module is indeed online at a given time. In case of ambiguity when looking for a module by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string containing either the serial number or the logical name of the desired module

**Returns :**

a `YModule` object allowing you to drive the module or get additional information on the module.

**YModule.FirstModule()****YModule****yFirstModule()YModule.FirstModule()**

Starts the enumeration of modules currently accessible.

```
def FirstModule( )
```

Use the method `YModule.nextModule()` to iterate on the next modules.

**Returns :**

a pointer to a `YModule` object, corresponding to the first module currently online, or a `null` pointer if there are none.

## module→describe()module.describe()

YModule

Returns a descriptive text that identifies the module.

```
def describe( )
```

The text may include either the logical name or the serial number of the module.

**Returns :**

a string that describes the module

**module→download()module.download()****YModule**

Downloads the specified built-in file and returns a binary buffer with its content.

```
def download( pathname)
```

**Parameters :**

**pathname** name of the new file to load

**Returns :**

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

## **module→functionCount()module.functionCount()**

**YModule**

Returns the number of functions (beside the "module" interface) available on the module.

```
def functionCount( )
```

**Returns :**

the number of functions on the module

On failure, throws an exception or returns a negative error code.

**module→functionId()module.functionId()****YModule**

Retrieves the hardware identifier of the *n*th function on the module.

```
def functionId( functionIndex)
```

**Parameters :**

**functionIndex** the index of the function for which the information is desired, starting at 0 for the first function.

**Returns :**

a string corresponding to the unambiguous hardware identifier of the requested module function

On failure, throws an exception or returns an empty string.

**module→functionName()module.functionName()****YModule**

Retrieves the logical name of the *n*th function on the module.

```
def functionName( functionIndex)
```

**Parameters :**

**functionIndex** the index of the function for which the information is desired, starting at 0 for the first function.

**Returns :**

a string corresponding to the logical name of the requested module function

On failure, throws an exception or returns an empty string.

**module→functionValue()module.functionValue()****YModule**

Retrieves the advertised value of the *n*th function on the module.

```
def functionValue( functionIndex)
```

**Parameters :**

**functionIndex** the index of the function for which the information is desired, starting at 0 for the first function.

**Returns :**

a short string (up to 6 characters) corresponding to the advertised value of the requested module function

On failure, throws an exception or returns an empty string.

**module→get\_beacon()**  
**module→beacon()module.get\_beacon()**

---

**YModule**

Returns the state of the localization beacon.

```
def get_beacon( )
```

**Returns :**

either Y\_BEACON\_OFF or Y\_BEACON\_ON, according to the state of the localization beacon

On failure, throws an exception or returns Y\_BEACON\_INVALID.

---

**module->getErrorMessage()****YModule****module->errorMessage()module.getErrorMessage()**

---

Returns the error message of the latest error with this module object.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using this module object

**module→get\_errorType()** **YModule**  
**module→errorType()module.get\_errorType()**

---

Returns the numerical error code of the latest error with this module object.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using this module object

**module->get\_firmwareRelease()**  
**module->firmwareRelease()**  
**module.get\_firmwareRelease()**

**YModule**

Returns the version of the firmware embedded in the module.

```
def get_firmwareRelease( )
```

**Returns :**

a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns Y\_FIRMWARERELEASE\_INVALID.

**module→get\_hardwareId()**  
**module→hardwareId()module.get\_hardwareId()**

---

**YModule**

Returns the unique hardware identifier of the module.

```
def get_hardwareId( )
```

The unique hardware identifier is made of the device serial number followed by string ".module".

**Returns :**

a string that uniquely identifies the module

**module→get\_icon2d()****YModule****module→icon2d()module.get\_icon2d()**

Returns the icon of the module.

```
def get_icon2d( )
```

The icon is a PNG image and does not exceeds 1536 bytes.

**Returns :**

a binary buffer with module icon, in png format.

**module→get\_lastLogs()**  
**module→lastLogs()module.get\_lastLogs()**

---

**YModule**

Returns a string with last logs of the module.

```
def get_lastLogs( )
```

This method return only logs that are still in the module.

**Returns :**

a string with last logs of the module.

---

<b>module-&gt;get_logicalName()</b>	<b>YModule</b>
<b>module-&gt;logicalName()module.get_logicalName()</b>	

---

Returns the logical name of the module.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the module

On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**module→get\_luminosity()**

**YModule**

**module→luminosity()module.get\_luminosity()**

---

Returns the luminosity of the module informative leds (from 0 to 100).

```
def get_luminosity( )
```

**Returns :**

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns Y\_LUMINOSITY\_INVALID.

**module→get\_persistentSettings()**  
**module→persistentSettings()**  
**module.get\_persistentSettings()**

**YModule**

Returns the current state of persistent module settings.

```
def get_persistentSettings( )
```

**Returns :**

a value among Y\_PERSISTENTSETTINGS\_LOADED, Y\_PERSISTENTSETTINGS\_SAVED and Y\_PERSISTENTSETTINGS\_MODIFIED corresponding to the current state of persistent module settings

On failure, throws an exception or returns Y\_PERSISTENTSETTINGS\_INVALID.

**module→get\_productId()** **YModule**  
**module→productId()module.get\_productId()**

---

Returns the USB device identifier of the module.

```
def get_productId( )
```

**Returns :**

an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns Y\_PRODUCTID\_INVALID.

---

<b>module-&gt;get_productName()</b>	<b>YModule</b>
<b>module-&gt;productName()module.get_productName()</b>	

---

Returns the commercial name of the module, as set by the factory.

```
def get_productName( )
```

**Returns :**

a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns Y\_PRODUCTNAME\_INVALID.

**module→get\_productRelease()**  
**module→productRelease()**  
**module.get\_productRelease()**

---

**YModule**

Returns the hardware release version of the module.

```
def get_productRelease( )
```

**Returns :**

an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns Y\_PRODUCTRELEASE\_INVALID.

---

**module->get\_rebootCountdown()**  
**module->rebootCountdown()**  
**module.get\_rebootCountdown()**

**YModule**

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

```
def get_rebootCountdown( )
```

**Returns :**

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

On failure, throws an exception or returns Y\_REBOOTCOUNTDOWN\_INVALID.

**module→get\_serialNumber()** **YModule**  
**module→serialNumber()module.get\_serialNumber()**

---

Returns the serial number of the module, as set by the factory.

```
def get_serialNumber( )
```

**Returns :**

a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns Y\_SERIALNUMBER\_INVALID.

---

<b>module-&gt;get_upTime()</b>	<b>YModule</b>
<b>module-&gt;upTime()module.get_upTime()</b>	

---

Returns the number of milliseconds spent since the module was powered on.

```
def get_upTime( )
```

**Returns :**

an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns Y\_UPTIME\_INVALID.

**module→get\_usbBandwidth()**  
**module→usbBandwidth()**  
**module.get\_usbBandwidth()**

**YModule**

Returns the number of USB interfaces used by the module.

```
def get_usbBandwidth( )
```

**Returns :**

either Y\_USBWIDTH\_SIMPLE or Y\_USBWIDTH\_DOUBLE, according to the number of USB interfaces used by the module

On failure, throws an exception or returns Y\_USBWIDTH\_INVALID.

**module→get\_usbCurrent()****YModule****module→usbCurrent()module.get\_usbCurrent()**

Returns the current consumed by the module on the USB bus, in milli-amps.

```
def get_usbCurrent( )
```

**Returns :**

an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns Y\_USBCURRENT\_INVALID.

**module→get(userData)**  
**module→userData()module.get(userData)**

---

**YModule**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**module→isOnline()|module.isOnline()****YModule**

Checks if the module is currently reachable, without raising any error.

```
def isOnline( )
```

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

**Returns :**

`true` if the module can be reached, and `false` otherwise

**module→load()module.load()****YModule**

Preloads the module cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded module parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**module→nextModule()module.nextModule()****YModule**

Continues the module enumeration started using `yFirstModule()`.

```
def nextModule( )
```

**Returns :**

a pointer to a `YModule` object, corresponding to the next module found, or a `null` pointer if there are no more modules to enumerate.

**module→reboot()|module.reboot()****YModule**

Schedules a simple module reboot after the given number of seconds.

```
def reboot( secBeforeReboot )
```

**Parameters :**

**secBeforeReboot** number of seconds before rebooting

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

---

**module→registerLogCallback()**  
**module.registerLogCallback()****YModule**

todo

```
def registerLogCallback( callback)
```

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments:  
the function object of which the value has changed, and the character string describing the new  
advertised value.

**module→revertFromFlash()**  
**module.revertFromFlash()**

**YModule**

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

```
def revertFromFlash( )
```

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**module→saveToFlash()module.saveToFlash()****YModule**

Saves current settings in the nonvolatile memory of the module.

```
def saveToFlash( )
```

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**module→set\_beacon()** **YModule**  
**module→setBeacon()module.set\_beacon()**

---

Turns on or off the module localization beacon.

```
def set_beacon( newval )
```

**Parameters :**

**newval** either Y\_BEACON\_OFF or Y\_BEACON\_ON

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

<b>module-&gt;set_logicalName()</b>	<b>YModule</b>
<b>module-&gt;setLogicalName()module.set_logicalName()</b>	

---

Changes the logical name of the module.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the module

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**module->set\_luminosity()** **YModule**  
**module->setLuminosity()module.set\_luminosity()**

---

Changes the luminosity of the module informative leds.

```
def set_luminosity( newval )
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** an integer corresponding to the luminosity of the module informative leds

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**module->set\_usbBandwidth()**  
**module->setUsbBandwidth()**  
**module.set\_usbBandwidth()**

**YModule**

Changes the number of USB interfaces used by the module.

```
def set_usbBandwidth( newval )
```

You must reboot the module after changing this setting.

**Parameters :**

**newval** either `Y_USBBANDWIDTH_SIMPLE` or `Y_USBBANDWIDTH_DOUBLE`, according to the number of USB interfaces used by the module

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**module→set(userData)** **YModule**  
**module→setUserData()****module.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

**module→triggerFirmwareUpdate()  
module.triggerFirmwareUpdate()****YModule**

Schedules a module reboot into special firmware update mode.

```
def triggerFirmwareUpdate( secBeforeReboot)
```

**Parameters :**

**secBeforeReboot** number of seconds before rebooting

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## 3.26. Network function interface

YNetwork objects provide access to TCP/IP parameters of Yoctopuce modules that include a built-in network interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_network.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YNetwork = yoctolib.YNetwork;
php	require_once('yocto_network.php');
cpp	#include "yocto_network.h"
m	#import "yocto_network.h"
pas	uses yocto_network;
vb	yocto_network.vb
cs	yocto_network.cs
java	import com.yoctopuce.YoctoAPI.YNetwork;
py	from yocto_network import *

### Global functions

#### yFindNetwork(func)

Retrieves a network interface for a given identifier.

#### yFirstNetwork()

Starts the enumeration of network interfaces currently accessible.

### YNetwork methods

#### network→callbackLogin(username, password)

Connects to the notification callback and saves the credentials required to log into it.

#### network→describe()

Returns a short text that describes unambiguously the instance of the network interface in the form TYPE ( NAME ) = SERIAL . FUNCTIONID.

#### network→get\_adminPassword()

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

#### network→get\_advertisedValue()

Returns the current value of the network interface (no more than 6 characters).

#### network→get\_callbackCredentials()

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

#### network→get\_callbackEncoding()

Returns the encoding standard to use for representing notification values.

#### network→get\_callbackMaxDelay()

Returns the maximum waiting time between two callback notifications, in seconds.

#### network→get\_callbackMethod()

Returns the HTTP method used to notify callbacks for significant state changes.

#### network→get\_callbackMinDelay()

Returns the minimum waiting time between two callback notifications, in seconds.

#### network→get\_callbackUrl()

Returns the callback URL to notify of significant state changes.

#### network→get\_discoverable()

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

**network→get\_errorMessage()**

Returns the error message of the latest error with the network interface.

**network→get\_errorType()**

Returns the numerical error code of the latest error with the network interface.

**network→get\_friendlyName()**

Returns a global identifier of the network interface in the format MODULE\_NAME . FUNCTION\_NAME.

**network→get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

**network→get\_functionId()**

Returns the hardware identifier of the network interface, without reference to the module.

**network→get\_hardwareId()**

Returns the unique hardware identifier of the network interface in the form SERIAL . FUNCTIONID.

**network→get\_ipAddress()**

Returns the IP address currently in use by the device.

**network→get\_logicalName()**

Returns the logical name of the network interface.

**network→get\_macAddress()**

Returns the MAC address of the network interface.

**network→get\_module()**

Gets the YModule object for the device on which the function is located.

**network→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**network→get\_poeCurrent()**

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

**network→get\_primaryDNS()**

Returns the IP address of the primary name server to be used by the module.

**network→get\_readiness()**

Returns the current established working mode of the network interface.

**network→get\_router()**

Returns the IP address of the router on the device subnet (default gateway).

**network→get\_secondaryDNS()**

Returns the IP address of the secondary name server to be used by the module.

**network→get\_subnetMask()**

Returns the subnet mask currently used by the device.

**network→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set(userData).

**network→get\_userPassword()**

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

**network→get\_wwwWatchdogDelay()**

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

**network→isOnline()**

Checks if the network interface is currently reachable, without raising any error.

**network→isOnline\_async(callback, context)**

Checks if the network interface is currently reachable, without raising any error (asynchronous version).

### 3. Reference

<b>network→load(msValidity)</b>	Preloads the network interface cache with a specified validity duration.
<b>network→load_async(msValidity, callback, context)</b>	Preloads the network interface cache with a specified validity duration (asynchronous version).
<b>network→nextNetwork()</b>	Continues the enumeration of network interfaces started using <code>yFirstNetwork()</code> .
<b>network→ping(host)</b>	Pings <code>str_host</code> to test the network connectivity.
<b>network→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>network→set_adminPassword(newval)</b>	Changes the password for the "admin" user.
<b>network→set_callbackCredentials(newval)</b>	Changes the credentials required to connect to the callback address.
<b>network→set_callbackEncoding(newval)</b>	Changes the encoding standard to use for representing notification values.
<b>network→set_callbackMaxDelay(newval)</b>	Changes the maximum waiting time between two callback notifications, in seconds.
<b>network→set_callbackMethod(newval)</b>	Changes the HTTP method used to notify callbacks for significant state changes.
<b>network→set_callbackMinDelay(newval)</b>	Changes the minimum waiting time between two callback notifications, in seconds.
<b>network→set_callbackUrl(newval)</b>	Changes the callback URL to notify significant state changes.
<b>network→set_discoverable(newval)</b>	Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).
<b>network→set_logicalName(newval)</b>	Changes the logical name of the network interface.
<b>network→set_primaryDNS(newval)</b>	Changes the IP address of the primary name server to be used by the module.
<b>network→set_secondaryDNS(newval)</b>	Changes the IP address of the secondary name server to be used by the module.
<b>network→set_userData(data)</b>	Stores a user context provided as argument in the <code>userData</code> attribute of the function.
<b>network→set_userPassword(newval)</b>	Changes the password for the "user" user.
<b>network→set_wwwWatchdogDelay(newval)</b>	Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.
<b>network→useDHCP(fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)</b>	Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.
<b>network→useStaticIP(ipAddress, subnetMaskLen, router)</b>	Changes the configuration of the network interface to use a static IP address.
<b>network→wait_async(callback, context)</b>	

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YNetwork.FindNetwork() yFindNetwork()YNetwork.FindNetwork()

**YNetwork**

Retrieves a network interface for a given identifier.

```
def FindNetwork( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the network interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YNetwork.isOnline()` to test if the network interface is indeed online at a given time. In case of ambiguity when looking for a network interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the network interface

### Returns :

a `YNetwork` object allowing you to drive the network interface.

**YNetwork.FirstNetwork()****YNetwork****yFirstNetwork()YNetwork.FirstNetwork()**

Starts the enumeration of network interfaces currently accessible.

```
def FirstNetwork( )
```

Use the method `YNetwork.nextNetwork()` to iterate on next network interfaces.

**Returns :**

a pointer to a `YNetwork` object, corresponding to the first network interface currently online, or a `null` pointer if there are none.

**network→callbackLogin()network.callbackLogin()****YNetwork**

Connects to the notification callback and saves the credentials required to log into it.

```
def callbackLogin( username, password)
```

The password is not stored into the module, only a hashed copy of the credentials are saved. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**username** username required to log to the callback

**password** password required to log to the callback

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→describe()network.describe()****YNetwork**

Returns a short text that describes unambiguously the instance of the network interface in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the network interface (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**network→get\_adminPassword()**  
**network→adminPassword()**  
**network.get\_adminPassword()**

**YNetwork**

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

```
def get_adminPassword( )
```

**Returns :**

a string corresponding to a hash string if a password has been set for user "admin", or an empty string otherwise

On failure, throws an exception or returns Y\_ADMINPASSWORD\_INVALID.

**network→get\_advertisedValue()**  
**network→advertisedValue()**  
**network.get\_advertisedValue()**

**YNetwork**

Returns the current value of the network interface (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the network interface (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**network→get\_callbackCredentials()**  
**network→callbackCredentials()**  
**network.get\_callbackCredentials()**

**YNetwork**

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

```
def get_callbackCredentials( )
```

**Returns :**

a string corresponding to a hashed version of the notification callback credentials if set, or an empty string otherwise

On failure, throws an exception or returns Y\_CALLBACKCREDENTIALS\_INVALID.

**network→get\_callbackEncoding()**  
**network→callbackEncoding()**  
**network.get\_callbackEncoding()**

**YNetwork**

Returns the encoding standard to use for representing notification values.

```
def get_callbackEncoding( )
```

**Returns :**

a value among Y\_CALLBACKENCODING\_FORM, Y\_CALLBACKENCODING\_JSON, Y\_CALLBACKENCODING\_JSON\_ARRAY, Y\_CALLBACKENCODING\_CSV and Y\_CALLBACKENCODING\_YOCTO\_API corresponding to the encoding standard to use for representing notification values

On failure, throws an exception or returns Y\_CALLBACKENCODING\_INVALID.

**network→get\_callbackMaxDelay()**  
**network→callbackMaxDelay()**  
**network.get\_callbackMaxDelay()**

**YNetwork**

Returns the maximum waiting time between two callback notifications, in seconds.

```
def get_callbackMaxDelay( )
```

**Returns :**

an integer corresponding to the maximum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns Y\_CALLBACKMAXDELAY\_INVALID.

**network→get\_callbackMethod()**  
**network→callbackMethod()**  
**network.get\_callbackMethod()**

**YNetwork**

Returns the HTTP method used to notify callbacks for significant state changes.

```
def get_callbackMethod( )
```

**Returns :**

a value among Y\_CALLBACKMETHOD\_POST, Y\_CALLBACKMETHOD\_GET and Y\_CALLBACKMETHOD\_PUT corresponding to the HTTP method used to notify callbacks for significant state changes

On failure, throws an exception or returns Y\_CALLBACKMETHOD\_INVALID.

**network→get\_callbackMinDelay()**  
**network→callbackMinDelay()**  
**network.get\_callbackMinDelay()**

---

**YNetwork**

Returns the minimum waiting time between two callback notifications, in seconds.

```
def get_callbackMinDelay( )
```

**Returns :**

an integer corresponding to the minimum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns Y\_CALLBACKMINDELAY\_INVALID.

---

**network→get\_callbackUrl()****YNetwork****network→callbackUrl()network.get\_callbackUrl()**

---

Returns the callback URL to notify of significant state changes.

```
def get_callbackUrl( )
```

**Returns :**

a string corresponding to the callback URL to notify of significant state changes

On failure, throws an exception or returns Y\_CALLBACKURL\_INVALID.

**network→get\_discoverable()** YNetwork  
**network→discoverable()network.get\_discoverable()**

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

```
def get_discoverable( )
```

**Returns :**

either Y\_DISCOVERABLE\_FALSE or Y\_DISCOVERABLE\_TRUE, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

On failure, throws an exception or returns Y\_DISCOVERABLE\_INVALID.

**network→get\_errorMessage()**  
**network→errorMessage()**  
**network.get\_errorMessage()**

**YNetwork**

Returns the error message of the latest error with the network interface.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the network interface object

**network→get\_errorType()**

**YNetwork**

**network→errorType()network.get\_errorType()**

---

Returns the numerical error code of the latest error with the network interface.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the network interface object

---

**network→get\_friendlyName()****YNetwork****network→friendlyName()network.get\_friendlyName()**

Returns a global identifier of the network interface in the format MODULE\_NAME.FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the network interface if they are defined, otherwise the serial number of the module and the hardware identifier of the network interface (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the network interface using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**network→get\_functionDescriptor()**  
**network→functionDescriptor()**  
**network.get\_functionDescriptor()**

**YNetwork**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

---

**network→get\_functionId()****YNetwork****network→functionId()network.get\_functionId()**

---

Returns the hardware identifier of the network interface, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the network interface (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

---

**network→get\_hardwareId()** YNetwork  
**network→hardwareId()network.get\_hardwareId()**

Returns the unique hardware identifier of the network interface in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the network interface. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the network interface (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

---

**network→get\_ipAddress()****YNetwork****network→ipAddress()network.get\_ipAddress()**

---

Returns the IP address currently in use by the device.

```
def get_ipAddress( )
```

The address may have been configured statically, or provided by a DHCP server.

**Returns :**

a string corresponding to the IP address currently in use by the device

On failure, throws an exception or returns Y\_IPADDRESS\_INVALID.

**network→get\_logicalName()**

**YNetwork**

**network→logicalName()network.get\_logicalName()**

---

Returns the logical name of the network interface.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the network interface. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

---

<b>network→get_macAddress()</b>	<b>YNetwork</b>
<b>network→macAddress()network.get_macAddress()</b>	

---

Returns the MAC address of the network interface.

```
def get_macAddress( )
```

The MAC address is also available on a sticker on the module, in both numeric and barcode forms.

**Returns :**

a string corresponding to the MAC address of the network interface

On failure, throws an exception or returns Y\_MACADDRESS\_INVALID.

**network→get\_module()**  
**network→module()network.get\_module()**

---

**YNetwork**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

---

**network→get\_poeCurrent()****YNetwork****network→poeCurrent()network.get\_poeCurrent()**

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

```
def get_poeCurrent( )
```

The current consumption is measured after converting PoE source to 5 Volt, and should never exceed 1800 mA.

**Returns :**

an integer corresponding to the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps

On failure, throws an exception or returns Y\_POECURRENT\_INVALID.

**network→get\_primaryDNS()**

**YNetwork**

**network→primaryDNS()network.get\_primaryDNS()**

---

Returns the IP address of the primary name server to be used by the module.

```
def get_primaryDNS( )
```

**Returns :**

a string corresponding to the IP address of the primary name server to be used by the module

On failure, throws an exception or returns Y\_PRIMARYDNS\_INVALID.

**network→get\_readiness()****YNetwork****network→readiness()network.get\_readiness()**

Returns the current established working mode of the network interface.

```
def get_readiness( )
```

Level zero (DOWN\_0) means that no hardware link has been detected. Either there is no signal on the network cable, or the selected wireless access point cannot be detected. Level 1 (LIVE\_1) is reached when the network is detected, but is not yet connected. For a wireless network, this shows that the requested SSID is present. Level 2 (LINK\_2) is reached when the hardware connection is established. For a wired network connection, level 2 means that the cable is attached at both ends. For a connection to a wireless access point, it shows that the security parameters are properly configured. For an ad-hoc wireless connection, it means that there is at least one other device connected on the ad-hoc network. Level 3 (DHCP\_3) is reached when an IP address has been obtained using DHCP. Level 4 (DNS\_4) is reached when the DNS server is reachable on the network. Level 5 (WWW\_5) is reached when global connectivity is demonstrated by properly loading the current time from an NTP server.

**Returns :**

a value among Y\_READINESS\_DOWN, Y\_READINESS\_EXISTS, Y\_READINESS\_LINKED, Y\_READINESS\_LAN\_OK and Y\_READINESS\_WWW\_OK corresponding to the current established working mode of the network interface

On failure, throws an exception or returns Y\_READINESS\_INVALID.

**network→get\_router()**

**YNetwork**

**network→router()network.get\_router()**

---

Returns the IP address of the router on the device subnet (default gateway).

```
def get_router( )
```

**Returns :**

a string corresponding to the IP address of the router on the device subnet (default gateway)

On failure, throws an exception or returns Y\_ROUTER\_INVALID.

**network→get\_secondaryDNS()**  
**network→secondaryDNS()**  
**network.get\_secondaryDNS()**

**YNetwork**

Returns the IP address of the secondary name server to be used by the module.

```
def get_secondaryDNS( )
```

**Returns :**

a string corresponding to the IP address of the secondary name server to be used by the module

On failure, throws an exception or returns Y\_SECONDARYDNS\_INVALID.

**network→get\_subnetMask()**

**YNetwork**

**network→subnetMask()network.get\_subnetMask()**

---

Returns the subnet mask currently used by the device.

```
def get_subnetMask( )
```

**Returns :**

a string corresponding to the subnet mask currently used by the device

On failure, throws an exception or returns Y\_SUBNETMASK\_INVALID.

---

**network→get(userData)****YNetwork****network→userData()network.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**network→get\_userPassword()**  
**network→userPassword()**  
**network.get\_userPassword()**

**YNetwork**

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

```
def get_userPassword( )
```

**Returns :**

a string corresponding to a hash string if a password has been set for "user" user, or an empty string otherwise

On failure, throws an exception or returns Y\_USERPASSWORD\_INVALID.

**network→get\_wwwWatchdogDelay()**  
**network→wwwWatchdogDelay()**  
**network.get\_wwwWatchdogDelay()**

**YNetwork**

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

```
def get_wwwWatchdogDelay( )
```

A zero value disables automated reboot in case of Internet connectivity loss.

**Returns :**

an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

On failure, throws an exception or returns Y\_WWWWATCHDOGDELAY\_INVALID.

**network→isOnline()network.isOnline()****YNetwork**

Checks if the network interface is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the network interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the network interface.

**Returns :**

`true` if the network interface can be reached, and `false` otherwise

**network→load()network.load()****YNetwork**

Preloads the network interface cache with a specified validity duration.

```
def load( msValidity )
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## network→nextNetwork()network.nextNetwork()

YNetwork

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

```
def nextNetwork( )
```

**Returns :**

a pointer to a YNetwork object, corresponding to a network interface currently online, or a null pointer if there are no more network interfaces to enumerate.

**network→ping()network.ping()****YNetwork**

Pings str\_host to test the network connectivity.

```
def ping( host)
```

Sends four ICMP ECHO\_REQUEST requests from the module to the target str\_host. This method returns a string with the result of the 4 ICMP ECHO\_REQUEST requests.

**Parameters :**

**host** the hostname or the IP address of the target

**Returns :**

a string with the result of the ping.

**network→registerValueCallback()**  
**network.registerValueCallback()****YNetwork**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**network→set\_adminPassword()**  
**network→setAdminPassword()**  
**network.set\_adminPassword()**

**YNetwork**

Changes the password for the "admin" user.

```
def set_adminPassword( newval)
```

This password becomes instantly required to perform any change of the module state. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the password for the "admin" user

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→set\_callbackCredentials()**  
**network→setCallbackCredentials()**  
**network.set\_callbackCredentials()**

**YNetwork**

Changes the credentials required to connect to the callback address.

```
def set_callbackCredentials( newval)
```

The credentials must be provided as returned by function `get_callbackCredentials`, in the form `username:hash`. The method used to compute the hash varies according to the authentication scheme implemented by the callback. For Basic authentication, the hash is the MD5 of the string `username:password`. For Digest authentication, the hash is the MD5 of the string `username:realm:password`. For a simpler way to configure callback credentials, use function `callbackLogin` instead. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the credentials required to connect to the callback address

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→set\_callbackEncoding()**  
**network→setCallbackEncoding()**  
**network.set\_callbackEncoding()**

**YNetwork**

Changes the encoding standard to use for representing notification values.

```
def set_callbackEncoding( newval)
```

**Parameters :**

**newval** a value among Y\_CALLBACKENCODING\_FORM, Y\_CALLBACKENCODING\_JSON, Y\_CALLBACKENCODING\_JSON\_ARRAY, Y\_CALLBACKENCODING\_CSV and Y\_CALLBACKENCODING\_YOCTO\_API corresponding to the encoding standard to use for representing notification values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→set\_callbackMaxDelay()**  
**network→setCallbackMaxDelay()**  
**network.set\_callbackMaxDelay()**

**YNetwork**

Changes the maximum waiting time between two callback notifications, in seconds.

```
def set_callbackMaxDelay( newval)
```

**Parameters :**

**newval** an integer corresponding to the maximum waiting time between two callback notifications, in seconds

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→set\_callbackMethod()**  
**network→setCallbackMethod()**  
**network.set\_callbackMethod()**

**YNetwork**

Changes the HTTP method used to notify callbacks for significant state changes.

```
def set_callbackMethod( newval)
```

**Parameters :**

**newval** a value among Y\_CALLBACKMETHOD\_POST, Y\_CALLBACKMETHOD\_GET and Y\_CALLBACKMETHOD\_PUT corresponding to the HTTP method used to notify callbacks for significant state changes

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→set\_callbackMinDelay()**  
**network→setCallbackMinDelay()**  
**network.set\_callbackMinDelay()**

**YNetwork**

Changes the minimum waiting time between two callback notifications, in seconds.

```
def set_callbackMinDelay( newval)
```

**Parameters :**

**newval** an integer corresponding to the minimum waiting time between two callback notifications, in seconds

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→set\_callbackUrl()****YNetwork****network→setCallbackUrl()network.set\_callbackUrl()**

Changes the callback URL to notify significant state changes.

```
def set_callbackUrl( newval)
```

Remember to call the saveToFlash( ) method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the callback URL to notify significant state changes

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→set\_discoverable()**  
**network→setDiscoverable()**  
**network.set\_discoverable()**

**YNetwork**

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

```
def set_discoverable( newval )
```

**Parameters :**

**newval** either `Y_DISCOVERABLE_FALSE` or `Y_DISCOVERABLE_TRUE`, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→set\_logicalName()**  
**network→setLogicalName()**  
**network.set\_logicalName()**

**YNetwork**

Changes the logical name of the network interface.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the network interface.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**network→set\_primaryDNS()** YNetwork  
**network→setPrimaryDNS()network.set\_primaryDNS()**

Changes the IP address of the primary name server to be used by the module.

```
def set_primaryDNS( newval)
```

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

**Parameters :**

**newval** a string corresponding to the IP address of the primary name server to be used by the module

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→set\_secondaryDNS()**  
**network→setSecondaryDNS()**  
**network.set\_secondaryDNS()**

**YNetwork**

Changes the IP address of the secondary name server to be used by the module.

```
def set_secondaryDNS( newval)
```

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the saveToFlash( ) method and then to reboot the module to apply this setting.

**Parameters :**

**newval** a string corresponding to the IP address of the secondary name server to be used by the module

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→set(userData)**

**YNetwork**

**network→setUserData()|network.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

**network→set\_userPassword()**  
**network→setUserPassword()**  
**network.set\_userPassword()**

**YNetwork**

Changes the password for the "user" user.

```
def set_userPassword( newval)
```

This password becomes instantly required to perform any use of the module. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the password for the "user" user

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→set\_wwwWatchdogDelay()**  
**network→setWwwWatchdogDelay()**  
**network.set\_wwwWatchdogDelay()**

**YNetwork**

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

```
def set_wwwWatchdogDelay( newval )
```

A zero value disables automated reboot in case of Internet connectivity loss. The smallest valid non-zero timeout is 90 seconds.

**Parameters :**

**newval** an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→useDHCP()network.useDHCP()****YNetwork**

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

```
def useDHCP( fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)
```

Until an address is received from a DHCP server, the module uses the IP parameters specified to this function. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

**Parameters :**

<b>fallbackIpAddr</b>	fallback IP address, to be used when no DHCP reply is received
<b>fallbackSubnetMaskLen</b>	fallback subnet mask length when no DHCP reply is received, as an integer (eg. 24 means 255.255.255.0)
<b>fallbackRouter</b>	fallback router IP address, to be used when no DHCP reply is received

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**network→useStaticIP()|network.useStaticIP()****YNetwork**

Changes the configuration of the network interface to use a static IP address.

```
def useStaticIP( ipAddress, subnetMaskLen, router)
```

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

**Parameters :**

**ipAddress** device IP address

**subnetMaskLen** subnet mask length, as an integer (eg. 24 means 255.255.255.0)

**router** router IP address (default gateway)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## 3.27. OS control

The OScontrol object allows some control over the operating system running a VirtualHub. OsControl is available on the VirtualHub software only. This feature must be activated at the VirtualHub start up with -o option.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_oscontrol.js'></script>
node.js	var yoctolib = require('yoctolib');
	var YOsControl = yoctolib.YOsControl;
php	require_once('yocto_oscontrol.php');
cpp	#include "yocto_oscontrol.h"
m	#import "yocto_oscontrol.h"
pas	uses yocto_oscontrol;
vb	yocto_oscontrol.vb
cs	yocto_oscontrol.cs
java	import com.yoctopuce.YoctoAPI.YOsControl;
py	from yocto_oscontrol import *

### Global functions

#### yFindOsControl(func)

Retrieves OS control for a given identifier.

#### yFirstOsControl()

Starts the enumeration of OS control currently accessible.

### YOsControl methods

#### oscontrol->describe()

Returns a short text that describes unambiguously the instance of the OS control in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

#### oscontrol->get\_advertisedValue()

Returns the current value of the OS control (no more than 6 characters).

#### oscontrol->get\_errorMessage()

Returns the error message of the latest error with the OS control.

#### oscontrol->get\_errorType()

Returns the numerical error code of the latest error with the OS control.

#### oscontrol->get\_friendlyName()

Returns a global identifier of the OS control in the format MODULE\_NAME . FUNCTION\_NAME.

#### oscontrol->get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### oscontrol->get\_functionId()

Returns the hardware identifier of the OS control, without reference to the module.

#### oscontrol->get\_hardwareId()

Returns the unique hardware identifier of the OS control in the form SERIAL . FUNCTIONID.

#### oscontrol->get\_logicalName()

Returns the logical name of the OS control.

#### oscontrol->get\_module()

Gets the YModule object for the device on which the function is located.

#### oscontrol->get\_module\_async(callback, context)

### 3. Reference

Gets the YModule object for the device on which the function is located (asynchronous version).

#### **oscontrol→get\_shutdownCountdown()**

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

#### **oscontrol→get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

#### **oscontrol→isOnline()**

Checks if the OS control is currently reachable, without raising any error.

#### **oscontrol→isOnline\_async(callback, context)**

Checks if the OS control is currently reachable, without raising any error (asynchronous version).

#### **oscontrol→load(msValidity)**

Preloads the OS control cache with a specified validity duration.

#### **oscontrol→load\_async(msValidity, callback, context)**

Preloads the OS control cache with a specified validity duration (asynchronous version).

#### **oscontrol→nextOsControl()**

Continues the enumeration of OS control started using `yFirstOsControl()`.

#### **oscontrol→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

#### **oscontrol→set\_logicalName(newval)**

Changes the logical name of the OS control.

#### **oscontrol→set(userData)**

Stores a user context provided as argument in the userData attribute of the function.

#### **oscontrol→shutdown(secBeforeShutDown)**

Schedules an OS shutdown after a given number of seconds.

#### **oscontrol→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YOsControl.FindOsControl() yFindOsControl()YOsControl.FindOsControl()

YOsControl

Retrieves OS control for a given identifier.

```
def FindOsControl( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the OS control is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YOsControl.isOnline()` to test if the OS control is indeed online at a given time. In case of ambiguity when looking for OS control by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the OS control

**Returns :**

a `YOsControl` object allowing you to drive the OS control.

## **YOsControl.FirstOsControl()**

**YOsControl**

### **yFirstOsControl()YOsControl.FirstOsControl()**

Starts the enumeration of OS control currently accessible.

```
def FirstOsControl( )
```

Use the method `YOsControl.nextOsControl()` to iterate on next OS control.

**Returns :**

a pointer to a `YOsControl` object, corresponding to the first OS control currently online, or a null pointer if there are none.

**oscontrol→describe()****YOscControl**

Returns a short text that describes unambiguously the instance of the OS control in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the OS control (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**oscontrol→get\_advertisedValue()**  
**oscontrol→advertisedValue()**  
**oscontrol.get\_advertisedValue()**

---

YOsControl

Returns the current value of the OS control (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the OS control (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

---

**oscontrol→get\_errorMessage()**  
**oscontrol→errorMessage()**  
**oscontrol.get\_errorMessage()**

**YOsControl**

Returns the error message of the latest error with the OS control.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the OS control object

**oscontrol→get\_errorType()**

**YOsControl**

**oscontrol→errorType()oscontrol.get\_errorType()**

---

Returns the numerical error code of the latest error with the OS control.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the OS control object

**oscontrol→get\_friendlyName()**  
**oscontrol→friendlyName()**  
**oscontrol.get\_friendlyName()**

**YOsControl**

Returns a global identifier of the OS control in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the OS control if they are defined, otherwise the serial number of the module and the hardware identifier of the OS control (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the OS control using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**oscontrol→get\_functionDescriptor()  
oscontrol→functionDescriptor()  
oscontrol.get\_functionDescriptor()****YOsControl**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**oscontrol→get\_functionId()****YOsControl****oscontrol→functionId()oscontrol.get\_functionId()**

Returns the hardware identifier of the OS control, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the OS control (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

<b>oscontrol→get_hardwareId()</b>	<b>YOsControl</b>
<b>oscontrol→hardwareId()oscontrol.get_hardwareId()</b>	

---

Returns the unique hardware identifier of the OS control in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the OS control. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the OS control (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**oscontrol→get\_logicalName()**  
**oscontrol→logicalName()**  
**oscontrol.get\_logicalName()**

**YOsControl**

Returns the logical name of the OS control.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the OS control. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**oscontrol→get\_module()**

**YOsControl**

**oscontrol→module()oscontrol.get\_module()**

---

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**oscontrol→get\_shutdownCountdown()**  
**oscontrol→shutdownCountdown()**  
**oscontrol.get\_shutdownCountdown()****YOsControl**

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

```
def get_shutdownCountdown( )
```

**Returns :**

an integer corresponding to the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled

On failure, throws an exception or returns Y\_SHUTDOWNCOUNTDOWN\_INVALID.

**oscontrol→get(userData)**

**YOsControl**

**oscontrol→userData()oscontrol.get(userData)**

---

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**oscontrol→isOnline()oscontrol.isOnline()****YOsControl**

Checks if the OS control is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the OS control in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the OS control.

**Returns :**

`true` if the OS control can be reached, and `false` otherwise

**oscontrol→load()oscontrol.load()****YOsControl**

Preloads the OS control cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

---

**oscontrol→nextOsControl()  
oscontrol.nextOsControl()****YOsControl**

Continues the enumeration of OS control started using `yFirstOsControl()`.

```
def nextOsControl( )
```

**Returns :**

a pointer to a `YOsControl` object, corresponding to OS control currently online, or a `null` pointer if there are no more OS control to enumerate.

**oscontrol→registerValueCallback()**  
**oscontrol.registerValueCallback()****YOsControl**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**oscontrol→set\_logicalName()**  
**oscontrol→setLogicalName()**  
**oscontrol.set\_logicalName()**

**YOsControl**

Changes the logical name of the OS control.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the OS control.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**oscontrol→set(userData)**

**YOsControl**

**oscontrol→setUserData()oscontrol.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData):
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

**oscontrol→shutdown()oscontrol.shutdown()****YOsControl**

Schedules an OS shutdown after a given number of seconds.

```
def shutdown( secBeforeShutDown)
```

**Parameters :**

**secBeforeShutDown** number of seconds before shutdown

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## 3.28. Power function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_power.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YPower = yoctolib.YPower;
php	require_once('yocto_power.php');
cpp	#include "yocto_power.h"
m	#import "yocto_power.h"
pas	uses yocto_power;
vb	yocto_power.vb
cs	yocto_power.cs
java	import com.yoctopuce.YoctoAPI.YPower;
py	from yocto_power import *

### Global functions

#### yFindPower(func)

Retrieves a electrical power sensor for a given identifier.

#### yFirstPower()

Starts the enumeration of electrical power sensors currently accessible.

### YPower methods

#### power->calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### power->describe()

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form TYPE ( NAME )=SERIAL . FUNCTIONID.

#### power->get\_advertisedValue()

Returns the current value of the electrical power sensor (no more than 6 characters).

#### power->get\_cosPhi()

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

#### power->get\_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### power->get\_currentValue()

Returns the current measure for the electrical power.

#### power->get\_errorMessage()

Returns the error message of the latest error with the electrical power sensor.

#### power->get\_errorType()

Returns the numerical error code of the latest error with the electrical power sensor.

#### power->get\_friendlyName()

Returns a global identifier of the electrical power sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### power->get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### power->get\_functionId()

Returns the hardware identifier of the electrical power sensor, without reference to the module.
<b>power→get_hardwareId()</b>
Returns the unique hardware identifier of the electrical power sensor in the form SERIAL . FUNCTIONID.
<b>power→get_highestValue()</b>
Returns the maximal value observed for the electrical power.
<b>power→get_logFrequency()</b>
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>power→get_logicalName()</b>
Returns the logical name of the electrical power sensor.
<b>power→get_lowestValue()</b>
Returns the minimal value observed for the electrical power.
<b>power→get_meter()</b>
Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.
<b>power→get_meterTimer()</b>
Returns the elapsed time since last energy counter reset, in seconds.
<b>power→get_module()</b>
Gets the YModule object for the device on which the function is located.
<b>power→get_module_async(callback, context)</b>
Gets the YModule object for the device on which the function is located (asynchronous version).
<b>power→get_recordedData(startTime, endTime)</b>
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>power→get_reportFrequency()</b>
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>power→get_resolution()</b>
Returns the resolution of the measured values.
<b>power→get_unit()</b>
Returns the measuring unit for the electrical power.
<b>power→get(userData)</b>
Returns the value of the userData attribute, as previously stored using method set(userData).
<b>power→isOnline()</b>
Checks if the electrical power sensor is currently reachable, without raising any error.
<b>power→isOnline_async(callback, context)</b>
Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).
<b>power→load(msValidity)</b>
Preloads the electrical power sensor cache with a specified validity duration.
<b>power→loadCalibrationPoints(rawValues, refValues)</b>
Retrieves error correction data points previously entered using the method calibrateFromPoints.
<b>power→load_async(msValidity, callback, context)</b>
Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).
<b>power→nextPower()</b>
Continues the enumeration of electrical power sensors started using yFirstPower( ).
<b>power→registerTimedReportCallback(callback)</b>
Registers the callback function that is invoked on every periodic timed notification.
<b>power→registerValueCallback(callback)</b>

### 3. Reference

---

Registers the callback function that is invoked on every change of advertised value.

**power→reset()**

Resets the energy counter.

**power→set\_highestValue(newval)**

Changes the recorded maximal value observed pour the electrical power.

**power→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**power→set\_logicalName(newval)**

Changes the logical name of the electrical power sensor.

**power→set\_lowestValue(newval)**

Changes the recorded minimal value observed pour the electrical power.

**power→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**power→set\_resolution(newval)**

Changes the resolution of the measured values.

**power→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**power→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**YPower.FindPower()****YPower****yFindPower()YPower.FindPower()**

Retrieves a electrical power sensor for a given identifier.

```
def FindPower( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the electrical power sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPower.isOnline()` to test if the electrical power sensor is indeed online at a given time. In case of ambiguity when looking for a electrical power sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the electrical power sensor

**Returns :**

a `YPower` object allowing you to drive the electrical power sensor.

## **YPower.FirstPower()**

**YPower**

### **yFirstPower()YPower.FirstPower()**

Starts the enumeration of electrical power sensors currently accessible.

```
def FirstPower( )
```

Use the method `YPower.nextPower()` to iterate on next electrical power sensors.

**Returns :**

a pointer to a `YPower` object, corresponding to the first electrical power sensor currently online, or a null pointer if there are none.

**power→calibrateFromPoints()**  
**power.calibrateFromPoints()****YPower**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**power→describe()power.describe()****YPower**

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

```
a string that describes the electrical power sensor (ex:  
Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)
```

**power→get\_advertisedValue()**  
**power→advertisedValue()**  
**power.get\_advertisedValue()**

**YPower**

Returns the current value of the electrical power sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the electrical power sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**power→get\_cosPhi()****YPower****power→cosPhi()power.get\_cosPhi()**

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

```
def get_cosPhi( )
```

**Returns :**

a floating point number corresponding to the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA)

On failure, throws an exception or returns Y\_COSPHI\_INVALID.

**power→get\_currentRawValue()**  
**power→currentRawValue()**  
**power.get\_currentRawValue()**

**YPower**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**power→get\_currentValue()** YPower  
**power→currentValue()power.get\_currentValue()**

---

Returns the current measure for the electrical power.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current measure for the electrical power

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

---

**power→getErrorMessage()****YPower****power→errorMessage()power.getErrorMessage()**

Returns the error message of the latest error with the electrical power sensor.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the electrical power sensor object

**power→get\_errorType()**  
**power→errorType()power.get\_errorType()****YPower**

Returns the numerical error code of the latest error with the electrical power sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the electrical power sensor object

---

<b>power→get_friendlyName()</b>	<b>YPower</b>
<b>power→friendlyName()power.get_friendlyName()</b>	

Returns a global identifier of the electrical power sensor in the format MODULE\_NAME.FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the electrical power sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the electrical power sensor (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the electrical power sensor using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**power→get\_functionDescriptor()**  
**power→functionDescriptor()**  
**power.get\_functionDescriptor()**

YPower

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

---

**power→get\_functionId()****YPower****power→functionId()power.get\_functionId()**

Returns the hardware identifier of the electrical power sensor, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the electrical power sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**power→get\_hardwareId()** YPower  
**power→hardwareId()power.get\_hardwareId()**

---

Returns the unique hardware identifier of the electrical power sensor in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the electrical power sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the electrical power sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**power→get\_highestValue()****YPower****power→highestValue()power.get\_highestValue()**

Returns the maximal value observed for the electrical power.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the electrical power

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**power→get\_logFrequency()** YPower  
**power→logFrequency()** `power.get_logFrequency()`

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**power→get\_logicalName()****YPower****power→logicalName()power.get\_logicalName()**

Returns the logical name of the electrical power sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the electrical power sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**power→get\_lowestValue()** YPower  
**power→lowestValue()power.get\_lowestValue()**

---

Returns the minimal value observed for the electrical power.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the electrical power

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**power→get\_meter()****YPower****power→meter()power.get\_meter()**

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

```
def get_meter( )
```

Note that this counter is reset at each start of the device.

**Returns :**

a floating point number corresponding to the energy counter, maintained by the wattmeter by integrating the power consumption over time

On failure, throws an exception or returns Y\_METER\_INVALID.

**power→get\_meterTimer()**

**YPower**

**power→meterTimer()power.get\_meterTimer()**

---

Returns the elapsed time since last energy counter reset, in seconds.

```
def get_meterTimer( )
```

**Returns :**

an integer corresponding to the elapsed time since last energy counter reset, in seconds

On failure, throws an exception or returns Y\_METERTIMER\_INVALID.

**power→get\_module()****YPower****power→module()power.get\_module()**

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

## power→get\_recordedData() power→recordedData()power.get\_recordedData()

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime )
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

### Parameters :

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

### Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**power→get\_reportFrequency()**  
**power→reportFrequency()**  
**power.get\_reportFrequency()**

**YPower**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**power→get\_resolution()**  
**power→resolution()power.get\_resolution()**

**YPower**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**power→get\_unit()****YPower****power→unit()power.get\_unit()**

Returns the measuring unit for the electrical power.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the electrical power

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**power→get(userData)**

**YPower**

**power→userData()power.get(userData())**

---

Returns the value of the userData attribute, as previously stored using method set(userData).

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**power→isOnline()power.isOnline()****YPower**

Checks if the electrical power sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the electrical power sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the electrical power sensor.

**Returns :**

`true` if the electrical power sensor can be reached, and `false` otherwise

**power→load()power.load()****YPower**

Preloads the electrical power sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**power→loadCalibrationPoints()**  
**power.loadCalibrationPoints()****YPower**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## **power→nextPower()power.nextPower()**

**YPower**

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

```
def nextPower( )
```

**Returns :**

a pointer to a `YPower` object, corresponding to a electrical power sensor currently online, or a null pointer if there are no more electrical power sensors to enumerate.

**power→registerTimedReportCallback()**  
**power.registerTimedReportCallback()****YPower**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**power→registerValueCallback()  
power.registerValueCallback()****YPower**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**power→reset()power.reset()****YPower**

Resets the energy counter.

```
def reset( )
```

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**power→set\_highestValue()** YPower  
**power→setHighestValue()** `power.set_highestValue()`

Changes the recorded maximal value observed pour the electrical power.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed pour the electrical power

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**power→set\_logFrequency()** YPower  
**power→setLogFrequency()power.set\_logFrequency()**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**power→set\_logicalName()** YPower  
**power→setLogicalName()power.set\_logicalName()**

---

Changes the logical name of the electrical power sensor.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the electrical power sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**power→set\_lowestValue()****YPower****power→setLowestValue()power.set\_lowestValue()**

Changes the recorded minimal value observed pour the electrical power.

```
def set_lowestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed pour the electrical power

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**power→set\_reportFrequency()**  
**power→setReportFrequency()**  
**power.set\_reportFrequency()**

YPower

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**power→set\_resolution()****YPower****power→setResolution()power.set\_resolution()**

Changes the resolution of the measured values.

```
def set_resolution( newval )
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**power→set(userData)** YPower  
**power→setUserData()****power.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.29. Pressure function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pressure.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YPressure = yoctolib.YPressure;
php	require_once('yocto_pressure.php');
cpp	#include "yocto_pressure.h"
m	#import "yocto_pressure.h"
pas	uses yocto_pressure;
vb	yocto_pressure.vb
cs	yocto_pressure.cs
java	import com.yoctopuce.YoctoAPI.YPressure;
py	from yocto_pressure import *

### Global functions

#### yFindPressure(func)

Retrieves a pressure sensor for a given identifier.

#### yFirstPressure()

Starts the enumeration of pressure sensors currently accessible.

### YPressure methods

#### pressure→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### pressure→describe()

Returns a short text that describes unambiguously the instance of the pressure sensor in the form TYPE (NAME )=SERIAL . FUNCTIONID.

#### pressure→get\_advertisedValue()

Returns the current value of the pressure sensor (no more than 6 characters).

#### pressure→get\_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

#### pressure→get\_currentValue()

Returns the current measure for the pressure.

#### pressure→get\_errorMessage()

Returns the error message of the latest error with the pressure sensor.

#### pressure→get\_errorType()

Returns the numerical error code of the latest error with the pressure sensor.

#### pressure→get\_friendlyName()

Returns a global identifier of the pressure sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### pressure→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### pressure→get\_functionId()

Returns the hardware identifier of the pressure sensor, without reference to the module.

#### pressure→get\_hardwareId()

Returns the unique hardware identifier of the pressure sensor in the form SERIAL . FUNCTIONID.

<b>pressure→get_highestValue()</b>	Returns the maximal value observed for the pressure.
<b>pressure→get_logFrequency()</b>	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>pressure→get_logicalName()</b>	Returns the logical name of the pressure sensor.
<b>pressure→get_lowestValue()</b>	Returns the minimal value observed for the pressure.
<b>pressure→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>pressure→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>pressure→get_recordedData(startTime, endTime)</b>	Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>pressure→get_reportFrequency()</b>	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>pressure→get_resolution()</b>	Returns the resolution of the measured values.
<b>pressure→get_unit()</b>	Returns the measuring unit for the pressure.
<b>pressure→get(userData)</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>pressure→isOnline()</b>	Checks if the pressure sensor is currently reachable, without raising any error.
<b>pressure→isOnline_async(callback, context)</b>	Checks if the pressure sensor is currently reachable, without raising any error (asynchronous version).
<b>pressure→load(msValidity)</b>	Preloads the pressure sensor cache with a specified validity duration.
<b>pressure→loadCalibrationPoints(rawValues, refValues)</b>	Retrieves error correction data points previously entered using the method calibrateFromPoints.
<b>pressure→load_async(msValidity, callback, context)</b>	Preloads the pressure sensor cache with a specified validity duration (asynchronous version).
<b>pressure→nextPressure()</b>	Continues the enumeration of pressure sensors started using yFirstPressure( ).
<b>pressure→registerTimedReportCallback(callback)</b>	Registers the callback function that is invoked on every periodic timed notification.
<b>pressure→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>pressure→set_highestValue(newval)</b>	Changes the recorded maximal value observed for the pressure.
<b>pressure→set_logFrequency(newval)</b>	Changes the datalogger recording frequency for this function.
<b>pressure→set_logicalName(newval)</b>	Changes the logical name of the pressure sensor.

**pressure→set\_lowestValue(newval)**

Changes the recorded minimal value observed for the pressure.

**pressure→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**pressure→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**pressure→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**pressure→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YPressure.FindPressure() yFindPressure()YPressure.FindPressure()

YPressure

Retrieves a pressure sensor for a given identifier.

```
def FindPressure( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the pressure sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPressure.isOnline()` to test if the pressure sensor is indeed online at a given time. In case of ambiguity when looking for a pressure sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the pressure sensor

### Returns :

a `YPressure` object allowing you to drive the pressure sensor.

**YPressure.FirstPressure()****YPressure****yFirstPressure()YPressure.FirstPressure()**

Starts the enumeration of pressure sensors currently accessible.

```
def FirstPressure( )
```

Use the method `YPressure.nextPressure( )` to iterate on next pressure sensors.

**Returns :**

a pointer to a `YPressure` object, corresponding to the first pressure sensor currently online, or a `null` pointer if there are none.

**pressure→calibrateFromPoints()****YPressure****pressure.calibrateFromPoints()**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**pressure→describe()pressure.describe()****YPressure**

Returns a short text that describes unambiguously the instance of the pressure sensor in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the pressure sensor (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**pressure→get\_advertisedValue()**  
**pressure→advertisedValue()**  
**pressure.get\_advertisedValue()**

**YPressure**

---

Returns the current value of the pressure sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the pressure sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**pressure→get\_currentRawValue()**  
**pressure→currentRawValue()**  
**pressure.get\_currentRawValue()**

**YPressure**

Returns the unrounded and uncalibrated raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**pressure→get\_currentValue()** YPressure  
**pressure→currentValue()pressure.get\_currentValue()**

---

Returns the current measure for the pressure.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current measure for the pressure

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

**pressure→get\_errorMessage()**  
**pressure→errorMessage()**  
**pressure.get\_errorMessage()**

**YPressure**

Returns the error message of the latest error with the pressure sensor.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the pressure sensor object

**pressure→get\_errorType()**

**YPressure**

**pressure→errorType()pressure.get\_errorType()**

---

Returns the numerical error code of the latest error with the pressure sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the pressure sensor object

**pressure→get\_friendlyName()**  
**pressure→friendlyName()**  
**pressure.get\_friendlyName()**

**YPressure**

Returns a global identifier of the pressure sensor in the format MODULE\_NAME . FUNCTION\_NAME.

**def get\_friendlyName( )**

The returned string uses the logical names of the module and of the pressure sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the pressure sensor (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the pressure sensor using logical names (ex: MyCustomName . relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

---

<b>pressure→get_functionDescriptor()</b>	<b>YPressure</b>
<b>pressure→functionDescriptor()</b>	
<b>pressure.get_functionDescriptor()</b>	

---

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**pressure→get\_functionId()**

**YPressure**

**pressure→functionId()pressure.get\_functionId()**

---

Returns the hardware identifier of the pressure sensor, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the pressure sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**pressure→get\_hwrid()**

**YPressure**

**pressure→hwrid()pressure.get\_hwrid()**

Returns the unique hardware identifier of the pressure sensor in the form SERIAL.FUNCTIONID.

```
def get_hwrid( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the pressure sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the pressure sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**pressure→get\_highestValue()**  
**pressure→highestValue()**  
**pressure.get\_highestValue()**

**YPressure**

Returns the maximal value observed for the pressure.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the pressure

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**pressure→get\_logFrequency()**  
**pressure→logFrequency()**  
**pressure.get\_logFrequency()**

**YPressure**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**pressure→get\_logicalName()**

**YPressure**

**pressure→logicalName()pressure.get\_logicalName()**

---

Returns the logical name of the pressure sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the pressure sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**pressure→get\_lowestValue()** YPressure  
**pressure→lowestValue()pressure.get\_lowestValue()**

---

Returns the minimal value observed for the pressure.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the pressure

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**pressure→get\_module()**

**YPressure**

**pressure→module()pressure.get\_module()**

---

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

**pressure→get\_recordedData()**  
**pressure→recordedData()**  
**pressure.get\_recordedData()**

**YPressure**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime )
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**pressure→get\_reportFrequency()**  
**pressure→reportFrequency()**  
**pressure.get\_reportFrequency()**

**YPressure**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**pressure→get\_resolution()**

**YPressure**

**pressure→resolution()pressure.get\_resolution()**

---

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**pressure→get\_unit()**

**YPressure**

**pressure→unit()pressure.get\_unit()**

Returns the measuring unit for the pressure.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the pressure

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**pressure→get(userData)**

**YPressure**

**pressure→userData()pressure.get(userData())**

---

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**pressure→isOnline()pressure.isOnline()****YPressure**

Checks if the pressure sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the pressure sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the pressure sensor.

**Returns :**

`true` if the pressure sensor can be reached, and `false` otherwise

**pressure→load()pressure.load()****YPressure**

Preloads the pressure sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**pressure→loadCalibrationPoints()  
pressure.loadCalibrationPoints()****YPressure**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## **pressure→nextPressure()pressure.nextPressure()**

**YPressure**

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

```
def nextPressure( )
```

**Returns :**

a pointer to a `YPressure` object, corresponding to a pressure sensor currently online, or a null pointer if there are no more pressure sensors to enumerate.

**pressure→registerTimedReportCallback()**  
**pressure.registerTimedReportCallback()****YPressure**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**pressure→registerValueCallback()  
pressure.registerValueCallback()****YPressure**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**pressure→set\_highestValue()**  
**pressure→setHighestValue()**  
**pressure.set\_highestValue()**

YPressure

Changes the recorded maximal value observed for the pressure.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed for the pressure

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**pressure→set\_logFrequency()**  
**pressure→setLogFrequency()**  
**pressure.set\_logFrequency()**

**YPressure**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**pressure→set\_logicalName()**  
**pressure→setLogicalName()**  
**pressure.set\_logicalName()**

**YPressure**

Changes the logical name of the pressure sensor.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the pressure sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**pressure→set\_lowestValue()**  
**pressure→setLowestValue()**  
**pressure.set\_lowestValue()**

**YPressure**

Changes the recorded minimal value observed for the pressure.

```
def set_lowestValue( newval )
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed for the pressure

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**pressure→set\_reportFrequency()**  
**pressure→setReportFrequency()**  
**pressure.set\_reportFrequency()**

**YPressure**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**pressure→set\_resolution()** YPressure  
**pressure→setResolution()pressure.set\_resolution()**

---

Changes the resolution of the measured physical values.

```
def set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**pressure→set(userData())**

**YPressure**

**pressure→setUserData()pressure.set(userData())**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.30. Pwm function interface

The Yoctopuce application programming interface allows you to configure, start, and stop the PWM.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_pwmoutput.js'></script>
nodejs var yoctolib = require('yoctolib');
var YPwmOutput = yoctolib.YPwmOutput;
require_once('yocto_pwmoutput.php');
php #include "yocto_pwmoutput.h"
cpp #import "yocto_pwmoutput.h"
m uses yocto_pwmoutput;
pas yocto_pwmoutput.vb
cs yocto_pwmoutput.cs
java import com.yoctopuce.YoctoAPI.YPwmOutput;
py from yocto_pwmoutput import *

```

### Global functions

#### **yFindPwmOutput(func)**

Retrieves a PWM for a given identifier.

#### **yFirstPwmOutput()**

Starts the enumeration of PWMs currently accessible.

### YPwmOutput methods

#### **pwmoutput→describe()**

Returns a short text that describes unambiguously the instance of the PWM in the form TYPE ( NAME ) = SERIAL . FUNCTIONID.

#### **pwmoutput→dutyCycleMove(target, ms\_duration)**

Performs a smooth change of the pulse duration toward a given value.

#### **pwmoutput→get\_advertisedValue()**

Returns the current value of the PWM (no more than 6 characters).

#### **pwmoutput→get\_dutyCycle()**

Returns the PWM duty cycle, in per cents.

#### **pwmoutput→get\_dutyCycleAtPowerOn()**

Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100

#### **pwmoutput→get\_enabled()**

Returns the state of the PWMs.

#### **pwmoutput→get\_enabledAtPowerOn()**

Returns the state of the PWM at device power on.

#### **pwmoutput→get\_errorMessage()**

Returns the error message of the latest error with the PWM.

#### **pwmoutput→get\_errorType()**

Returns the numerical error code of the latest error with the PWM.

#### **pwmoutput→get\_frequency()**

Returns the PWM frequency in Hz.

#### **pwmoutput→get\_friendlyName()**

Returns a global identifier of the PWM in the format MODULE\_NAME . FUNCTION\_NAME.

#### **pwmoutput→get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

**pwmoutput→get\_functionId()**

Returns the hardware identifier of the PWM, without reference to the module.

**pwmoutput→get\_hardwareId()**

Returns the unique hardware identifier of the PWM in the form SERIAL.FUNCTIONID.

**pwmoutput→get\_logicalName()**

Returns the logical name of the PWM.

**pwmoutput→get\_module()**

Gets the YModule object for the device on which the function is located.

**pwmoutput→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**pwmoutput→get\_period()**

Returns the PWM period in milliseconds.

**pwmoutput→get\_pulseDuration()**

Returns the PWM pulse length in milliseconds.

**pwmoutput→get(userData)**

Returns the value of the userData attribute, as previously stored using method set(userData).

**pwmoutput→isOnline()**

Checks if the PWM is currently reachable, without raising any error.

**pwmoutput→isOnline\_async(callback, context)**

Checks if the PWM is currently reachable, without raising any error (asynchronous version).

**pwmoutput→load(msValidity)**

Preloads the PWM cache with a specified validity duration.

**pwmoutput→load\_async(msValidity, callback, context)**

Preloads the PWM cache with a specified validity duration (asynchronous version).

**pwmoutput→nextPwmOutput()**

Continues the enumeration of PWMs started using yFirstPwmOutput( ).

**pwmoutput→pulseDurationMove(ms\_target, ms\_duration)**

Performs a smooth transition of the pulse duration toward a given value.

**pwmoutput→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**pwmoutput→set\_dutyCycle(newval)**

Changes the PWM duty cycle, in per cents.

**pwmoutput→set\_dutyCycleAtPowerOn(newval)**

Changes the PWM duty cycle at device power on.

**pwmoutput→set\_enabled(newval)**

Stops or starts the PWM.

**pwmoutput→set\_enabledAtPowerOn(newval)**

Changes the state of the PWM at device power on.

**pwmoutput→set\_frequency(newval)**

Changes the PWM frequency.

**pwmoutput→set\_logicalName(newval)**

Changes the logical name of the PWM.

**pwmoutput→set\_period(newval)**

Changes the PWM period.

### **3. Reference**

---

**pwmoutput→set\_pulseDuration(newval)**

Changes the PWM pulse length, in milliseconds.

**pwmoutput→set(userData)**

Stores a user context provided as argument in the userData attribute of the function.

**pwmoutput→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YPwmOutput.FindPwmOutput() yFindPwmOutput()YPwmOutput.FindPwmOutput()

YPwmOutput

Retrieves a PWM for a given identifier.

```
def FindPwmOutput( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the PWM is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmOutput.isOnline()` to test if the PWM is indeed online at a given time. In case of ambiguity when looking for a PWM by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the PWM

### Returns :

a `YPwmOutput` object allowing you to drive the PWM.

## YPwmOutput.FirstPwmOutput()

**YPwmOutput**

### yFirstPwmOutput()YPwmOutput.FirstPwmOutput()

---

Starts the enumeration of PWMs currently accessible.

```
def FirstPwmOutput( )
```

Use the method `YPwmOutput.nextPwmOutput()` to iterate on next PWMs.

**Returns :**

a pointer to a `YPwmOutput` object, corresponding to the first PWM currently online, or a `null` pointer if there are none.

**pwmoutput→describe()pwmoutput.describe()****YPwmOutput**

Returns a short text that describes unambiguously the instance of the PWM in the form TYPE (NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the PWM (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**pwmoutput→dutyCycleMove()**  
**pwmoutput.dutyCycleMove()****YPwmOutput**

Performs a smooth change of the pulse duration toward a given value.

```
def dutyCycleMove( target, ms_duration)
```

**Parameters :**

**target** new duty cycle at the end of the transition (floating-point number, between 0 and 1)

**ms\_duration** total duration of the transition, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**pwmoutput→get\_advertisedValue()**  
**pwmoutput→advertisedValue()**  
**pwmoutput.get\_advertisedValue()**

**YPwmOutput**

Returns the current value of the PWM (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the PWM (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**pwmoutput→get\_dutyCycle()**

**YPwmOutput**

**pwmoutput→dutyCycle()pwmoutput.get\_dutyCycle()**

---

Returns the PWM duty cycle, in per cents.

```
def get_dutyCycle( )
```

**Returns :**

a floating point number corresponding to the PWM duty cycle, in per cents

On failure, throws an exception or returns Y\_DUTYCYCLE\_INVALID.

`pwmoutput->get_dutyCycleAtPowerOn()`  
`pwmoutput->dutyCycleAtPowerOn()`  
`pwmoutput.get_dutyCycleAtPowerOn()`

`YPwmOutput`

Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100

```
def get_dutyCycleAtPowerOn( )
```

**Returns :**

a floating point number corresponding to the PWMs duty cycle at device power on as a floating point number between 0 and 100

On failure, throws an exception or returns `Y_DUTYCYLEATPOWERON_INVALID`.

**pwmoutput→get\_enabled()**

**YPwmOutput**

**pwmoutput→enabled()pwmoutput.get\_enabled()**

---

Returns the state of the PWMs.

```
def get_enabled( )
```

**Returns :**

either Y\_ENABLED\_FALSE or Y\_ENABLED\_TRUE, according to the state of the PWMs

On failure, throws an exception or returns Y\_ENABLED\_INVALID.

**pwmoutput→get\_enabledAtPowerOn()**  
**pwmoutput→enabledAtPowerOn()**  
**pwmoutput.get\_enabledAtPowerOn()**

**YPwmOutput**

Returns the state of the PWM at device power on.

```
def get_enabledAtPowerOn( )
```

**Returns :**

either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the state of the PWM at device power on

On failure, throws an exception or returns `Y_ENABLEDATPOWERON_INVALID`.

**pwmoutput→get\_errorMessage()**  
**pwmoutput→errorMessage()**  
**pwmoutput.get\_errorMessage()**

**YPwmOutput**

Returns the error message of the latest error with the PWM.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the PWM object

---

**pwmoutput→get\_errorType()****YPwmOutput****pwmoutput→errorType()pwmoutput.get\_errorType()**

---

Returns the numerical error code of the latest error with the PWM.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the PWM object

**pwmoutput→get\_frequency()**

**YPwmOutput**

**pwmoutput→frequency()pwmoutput.get\_frequency()**

---

Returns the PWM frequency in Hz.

```
def get_frequency( )
```

**Returns :**

an integer corresponding to the PWM frequency in Hz

On failure, throws an exception or returns Y\_FREQUENCY\_INVALID.

**pwmoutput→get\_friendlyName()**  
**pwmoutput→friendlyName()**  
**pwmoutput.get\_friendlyName()**

**YPwmOutput**

Returns a global identifier of the PWM in the format MODULE\_NAME . FUNCTION\_NAME.

**def get\_friendlyName( )**

The returned string uses the logical names of the module and of the PWM if they are defined, otherwise the serial number of the module and the hardware identifier of the PWM (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the PWM using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**pwmoutput→get\_functionDescriptor()**  
**pwmoutput→functionDescriptor()**  
**pwmoutput.get\_functionDescriptor()**

**YPwmOutput**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**pwmoutput→get\_functionId()****YPwmOutput****pwmoutput→functionId()pwmoutput.get\_functionId()**

Returns the hardware identifier of the PWM, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the PWM (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**pwmoutput→get\_hardwareId()**  
**pwmoutput→hardwareId()**  
**pwmoutput.get\_hardwareId()**

**YPwmOutput**

Returns the unique hardware identifier of the PWM in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the PWM. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the PWM (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

`pwmoutput→get_logicalName()`  
`pwmoutput→logicalName()`  
`pwmoutput.get_logicalName()`

`YPwmOutput`

Returns the logical name of the PWM.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the PWM. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

**pwmoutput→get\_module()****YPwmOutput****pwmoutput→module()pwmoutput.get\_module()**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**pwmoutput→get\_period()****YPwmOutput****pwmoutput→period()pwmoutput.get\_period()**

Returns the PWM period in milliseconds.

```
def get_period( )
```

**Returns :**

a floating point number corresponding to the PWM period in milliseconds

On failure, throws an exception or returns Y\_PERIOD\_INVALID.

**pwmoutput→get\_pulseDuration()**  
**pwmoutput→pulseDuration()**  
**pwmoutput.get\_pulseDuration()**

---

**YPwmOutput**

Returns the PWM pulse length in milliseconds.

```
def get_pulseDuration( )
```

**Returns :**

a floating point number corresponding to the PWM pulse length in milliseconds

On failure, throws an exception or returns Y\_PULSEDURATION\_INVALID.

---

**pwmoutput→get(userData)****YPwmOutput****pwmoutput→userData()pwmoutput.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**pwmoutput→isOnline()pwmoutput.isOnline()****YPwmOutput**

Checks if the PWM is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the PWM in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the PWM.

**Returns :**

`true` if the PWM can be reached, and `false` otherwise

**pwmoutput→load()pwmoutput.load()****YPwmOutput**

Preloads the PWM cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**pwmoutput→nextPwmOutput()**  
**pwmoutput.nextPwmOutput()**

---

**YPwmOutput**

Continues the enumeration of PWMs started using `yFirstPwmOutput( )`.

```
def nextPwmOutput( )
```

**Returns :**

a pointer to a `YPwmOutput` object, corresponding to a PWM currently online, or a `null` pointer if there are no more PWMs to enumerate.

**pwmoutput→pulseDurationMove()**  
**pwmoutput.pulseDurationMove()****YPwmOutput**

Performs a smooth transition of the pulse duration toward a given value.

```
def pulseDurationMove( ms_target, ms_duration)
```

Any period, frequency, duty cycle or pulse width change will cancel any ongoing transition process.

**Parameters :**

**ms\_target** new pulse duration at the end of the transition (floating-point number, representing the pulse duration in milliseconds)

**ms\_duration** total duration of the transition, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**pwmoutput→registerValueCallback()  
pwmoutput.registerValueCallback()****YPwmOutput**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

`pwmoutput->set_dutyCycle()`  
`pwmoutput->setDutyCycle()`  
`pwmoutput.set_dutyCycle()`

YPwmOutput

Changes the PWM duty cycle, in per cents.

```
def set_dutyCycle( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the PWM duty cycle, in per cents

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**pwmoutput→set\_dutyCycleAtPowerOn()**  
**pwmoutput→setDutyCycleAtPowerOn()**  
**pwmoutput.set\_dutyCycleAtPowerOn()**

**YPwmOutput**

Changes the PWM duty cycle at device power on.

```
def set_dutyCycleAtPowerOn( newval)
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters :**

**newval** a floating point number corresponding to the PWM duty cycle at device power on

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

`pwmoutput->set_enabled()`

`YPwmOutput`

`pwmoutput->setEnabled()pwmoutput.set_enabled()`

Stops or starts the PWM.

```
def set_enabled( newval)
```

**Parameters :**

`newval` either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**pwmoutput→set\_enabledAtPowerOn()**  
**pwmoutput→setEnabledAtPowerOn()**  
**pwmoutput.set\_enabledAtPowerOn()**

**YPwmOutput**

Changes the state of the PWM at device power on.

```
def set_enabledAtPowerOn( newval)
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters :**

**newval** either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the state of the PWM at device power on

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**pwmoutput→set\_frequency()**  
**pwmoutput→setFrequency()**  
**pwmoutput.set\_frequency()**

**YPwmOutput**

Changes the PWM frequency.

```
def set_frequency( newval )
```

The duty cycle is kept unchanged thanks to an automatic pulse width change.

**Parameters :**

**newval** an integer corresponding to the PWM frequency

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**pwmoutput→set\_logicalName()**  
**pwmoutput→setLogicalName()**  
**pwmoutput.set\_logicalName()**

**YPwmOutput**

Changes the logical name of the PWM.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the PWM.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**pwmoutput→set\_period()**

**YPwmOutput**

**pwmoutput→setPeriod()pwmoutput.set\_period()**

Changes the PWM period.

```
def set_period( newval )
```

**Parameters :**

**newval** a floating point number corresponding to the PWM period

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

<b>pwmoutput→set_pulseDuration()</b>	<b>YPwmOutput</b>
<b>pwmoutput→setPulseDuration()</b>	
<b>pwmoutput.set_pulseDuration()</b>	

---

Changes the PWM pulse length, in milliseconds.

```
def set_pulseDuration( newval)
```

A pulse length cannot be longer than period, otherwise it is truncated.

**Parameters :**

**newval** a floating point number corresponding to the PWM pulse length, in milliseconds

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**pwmoutput→set(userData)**  
**pwmoutput→setUserData()**  
**pwmoutput.set(userData)**

**YPwmOutput**

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData) data
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.31. PwmPowerSource function interface

The Yoctopuce application programming interface allows you to configure the voltage source used by all PWM on the same device.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_pwmpowersource.js'></script>
nodejs var yoctolib = require('yoctolib');
var YPwmPowerSource = yoctolib.YPwmPowerSource;
php require_once('yocto_pwmpowersource.php');
cpp #include "yocto_pwmpowersource.h"
m #import "yocto_pwmpowersource.h"
pas uses yocto_pwmpowersource;
vb yocto_pwmpowersource.vb
cs yocto_pwmpowersource.cs
java import com.yoctopuce.YoctoAPI.YPwmPowerSource;
py from yocto_pwmpowersource import *

```

### Global functions

#### **yFindPwmPowerSource(func)**

Retrieves a voltage source for a given identifier.

#### **yFirstPwmPowerSource()**

Starts the enumeration of Voltage sources currently accessible.

### YPwmPowerSource methods

#### **pwmpowersource→describe()**

Returns a short text that describes unambiguously the instance of the voltage source in the form  
TYPE ( NAME ) = SERIAL . FUNCTIONID.

#### **pwmpowersource→get\_advertisedValue()**

Returns the current value of the voltage source (no more than 6 characters).

#### **pwmpowersource→get\_errorMessage()**

Returns the error message of the latest error with the voltage source.

#### **pwmpowersource→get\_errorType()**

Returns the numerical error code of the latest error with the voltage source.

#### **pwmpowersource→get\_friendlyName()**

Returns a global identifier of the voltage source in the format MODULE\_NAME . FUNCTION\_NAME.

#### **pwmpowersource→get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### **pwmpowersource→get\_functionId()**

Returns the hardware identifier of the voltage source, without reference to the module.

#### **pwmpowersource→get\_hardwareId()**

Returns the unique hardware identifier of the voltage source in the form SERIAL . FUNCTIONID.

#### **pwmpowersource→get\_logicalName()**

Returns the logical name of the voltage source.

#### **pwmpowersource→get\_module()**

Gets the YModule object for the device on which the function is located.

#### **pwmpowersource→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**pwmpowersource→get\_powerMode()**

Returns the selected power source for the PWM on the same device

**pwmpowersource→get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

**pwmpowersource→isOnline()**

Checks if the voltage source is currently reachable, without raising any error.

**pwmpowersource→isOnline\_async(callback, context)**

Checks if the voltage source is currently reachable, without raising any error (asynchronous version).

**pwmpowersource→load(msValidity)**

Preloads the voltage source cache with a specified validity duration.

**pwmpowersource→load\_async(msValidity, callback, context)**

Preloads the voltage source cache with a specified validity duration (asynchronous version).

**pwmpowersource→nextPwmPowerSource()**

Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

**pwmpowersource→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**pwmpowersource→set\_logicalName(newval)**

Changes the logical name of the voltage source.

**pwmpowersource→set\_powerMode(newval)**

Changes the PWM power source.

**pwmpowersource→set(userData)**

Stores a user context provided as argument in the userData attribute of the function.

**pwmpowersource→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**YPwmPowerSource.FindPwmPowerSource()**  
**yFindPwmPowerSource()**  
**YPwmPowerSource.FindPwmPowerSource()**

**YPwmPowerSource**

Retrieves a voltage source for a given identifier.

```
def FindPwmPowerSource( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage source is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPwmPowerSource.isOnline()` to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the voltage source

**Returns :**

a `YPwmPowerSource` object allowing you to drive the voltage source.

**YPwmPowerSource.FirstPwmPowerSource()**  
**yFirstPwmPowerSource()**  
**YPwmPowerSource.FirstPwmPowerSource()**

**YPwmPowerSource**

Starts the enumeration of Voltage sources currently accessible.

```
def FirstPwmPowerSource( )
```

Use the method `YPwmPowerSource.nextPwmPowerSource()` to iterate on next Voltage sources.

**Returns :**

a pointer to a `YPwmPowerSource` object, corresponding to the first source currently online, or a null pointer if there are none.

**pwmpowersource→describe()**  
**pwmpowersource.describe()****YPwmPowerSource**

Returns a short text that describes unambiguously the instance of the voltage source in the form  
TYPE ( NAME )=SERIAL . FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the voltage source (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**pwmpowersource→get\_advertisedValue()**  
**pwmpowersource→advertisedValue()**  
**pwmpowersource.get\_advertisedValue()**

**YPwmPowerSource**

Returns the current value of the voltage source (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the voltage source (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**pwmpowersource→get\_errorMessage()**  
**pwmpowersource→errorMessage()**  
**pwmpowersource.get\_errorMessage()**

**YPwmPowerSource**

---

Returns the error message of the latest error with the voltage source.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the voltage source object

**pwmpowersource→get\_errorType()**  
**pwmpowersource→errorType()**  
**pwmpowersource.get\_errorType()**

**YPwmPowerSource**

Returns the numerical error code of the latest error with the voltage source.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the voltage source object

**pwmpowersource→get\_friendlyName()**  
**pwmpowersource→friendlyName()**  
**pwmpowersource.get\_friendlyName()**

**YPwmPowerSource**

Returns a global identifier of the voltage source in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the voltage source if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage source (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the voltage source using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**pwmpowersource→get\_functionDescriptor()**  
**pwmpowersource→functionDescriptor()**  
**pwmpowersource.get\_functionDescriptor()**

**YPwmPowerSource**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**pwmpowersource→get\_functionId()**  
**pwmpowersource→functionId()**  
**pwmpowersource.get\_functionId()**

**YPwmPowerSource**

Returns the hardware identifier of the voltage source, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the voltage source (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**pwmpowersource→get\_hardwareId()**  
**pwmpowersource→hardwareId()**  
**pwmpowersource.get\_hardwareId()**

**YPwmPowerSource**

Returns the unique hardware identifier of the voltage source in the form SERIAL.FUNCTIONID.

**def get\_hardwareId( )**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage source. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the voltage source (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**pwmpowersource→get\_logicalName()**  
**pwmpowersource→logicalName()**  
**pwmpowersource.get\_logicalName()**

**YPwmPowerSource**

Returns the logical name of the voltage source.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the voltage source. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**pwmpowersource→get\_module()**  
**pwmpowersource→module()**  
**pwmpowersource.get\_module()**

**YPwmPowerSource**

Gets the **YModule** object for the device on which the function is located.

**def get\_module( )**

If the function cannot be located on any module, the returned instance of **YModule** is not shown as online.

**Returns :**

an instance of **YModule**

**pwmpowersource→get\_powerMode()**  
**pwmpowersource→powerMode()**  
**pwmpowersource.get\_powerMode()**

**YPwmPowerSource**

Returns the selected power source for the PWM on the same device

```
def get_powerMode( )
```

**Returns :**

a value among Y\_POWERMODE\_USB\_5V, Y\_POWERMODE\_USB\_3V, Y\_POWERMODE\_EXT\_V and  
Y\_POWERMODE\_OPNDRN corresponding to the selected power source for the PWM on the same device

On failure, throws an exception or returns Y\_POWERMODE\_INVALID.

**pwmpowersource→get(userData)**  
**pwmpowersource→userData()**  
**pwmpowersource.get(userData)**

**YPwmPowerSource**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

`def get(userData) :`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**pwmpowersource→isOnline()**  
**pwmpowersource.isOnline()**

---

**YPwmPowerSource**

Checks if the voltage source is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the voltage source in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage source.

**Returns :**

true if the voltage source can be reached, and false otherwise

**pwmpowersource→load()pwmpowersource.load()****YPwmPowerSource**

Preloads the voltage source cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**pwmpowersource→nextPwmPowerSource()**  
**pwmpowersource.nextPwmPowerSource()**

---

**YPwmPowerSource**

Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

```
def nextPwmPowerSource( )
```

**Returns :**

a pointer to a `YPwmPowerSource` object, corresponding to a voltage source currently online, or a null pointer if there are no more Voltage sources to enumerate.

**pwmpowersource→registerValueCallback()**  
**pwmpowersource.registerValueCallback()****YPwmPowerSource**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**pwmpowersource→set\_logicalName()**  
**pwmpowersource→setLogicalName()**  
**pwmpowersource.set\_logicalName()**

**YPwmPowerSource**

Changes the logical name of the voltage source.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the voltage source.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**pwmpowersource→set\_powerMode()**  
**pwmpowersource→setPowerMode()**  
**pwmpowersource.set\_powerMode()**

**YPwmPowerSource**

Changes the PWM power source.

```
def set_powerMode( newval)
```

PWM can use isolated 5V from USB, isolated 3V from USB or voltage from an external power source. The PWM can also work in open drain mode. In that mode, the PWM actively pulls the line down. Warning: this setting is common to all PWM on the same device. If you change that parameter, all PWM located on the same device are affected. If you want the change to be kept after a device reboot, make sure to call the matching module saveToFlash( ).

**Parameters :**

**newval** a value among Y\_POWERMODE\_USB\_5V, Y\_POWERMODE\_USB\_3V, Y\_POWERMODE\_EXT\_V and Y\_POWERMODE\_OPNDRN corresponding to the PWM power source

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**pwmpowersource→set(userData)**  
**pwmpowersource→setUserData()**  
**pwmpowersource.set(userData)**

**YPwmPowerSource**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.32. Quaternion interface

The Yoctopuce API YQt class provides direct access to the Yocto3D attitude estimation using a quaternion. It is usually not needed to use the YQt class directly, as the YGyro class provides a more convenient higher-level interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_gyro.js'></script>
node.js	var yoctolib = require('yoctolib');
	var YGyro = yoctolib.YGyro;
php	require_once('yocto_gyro.php');
cpp	#include "yocto_gyro.h"
m	#import "yocto_gyro.h"
pas	uses yocto_gyro;
vb	yocto_gyro.vb
cs	yocto_gyro.cs
java	import com.yoctopuce.YoctoAPI.YGyro;
py	from yocto_gyro import *

### Global functions

#### yFindQt(func)

Retrieves a quaternion component for a given identifier.

#### yFirstQt()

Starts the enumeration of quaternion components currently accessible.

### YQt methods

#### qt→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### qt→describe()

Returns a short text that describes unambiguously the instance of the quaternion component in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

#### qt→get\_advertisedValue()

Returns the current value of the quaternion component (no more than 6 characters).

#### qt→get\_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### qt→get\_currentValue()

Returns the current value of the value.

#### qt→get\_errorMessage()

Returns the error message of the latest error with the quaternion component.

#### qt→get\_errorType()

Returns the numerical error code of the latest error with the quaternion component.

#### qt→get\_friendlyName()

Returns a global identifier of the quaternion component in the format MODULE\_NAME . FUNCTION\_NAME.

#### qt→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### qt→get\_functionId()

Returns the hardware identifier of the quaternion component, without reference to the module.

#### qt→get\_hardwareId()

Returns the unique hardware identifier of the quaternion component in the form SERIAL . FUNCTIONID.
<b>qt→get_highestValue()</b>
Returns the maximal value observed for the value since the device was started.
<b>qt→get_logFrequency()</b>
Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>qt→get_logicalName()</b>
Returns the logical name of the quaternion component.
<b>qt→get_lowestValue()</b>
Returns the minimal value observed for the value since the device was started.
<b>qt→get_module()</b>
Gets the YModule object for the device on which the function is located.
<b>qt→get_module_async(callback, context)</b>
Gets the YModule object for the device on which the function is located (asynchronous version).
<b>qt→get_recordedData(startTime, endTime)</b>
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>qt→get_reportFrequency()</b>
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>qt→get_resolution()</b>
Returns the resolution of the measured values.
<b>qt→get_unit()</b>
Returns the measuring unit for the value.
<b>qt→get(userData)</b>
Returns the value of the userData attribute, as previously stored using method set(userData).
<b>qt→isOnline()</b>
Checks if the quaternion component is currently reachable, without raising any error.
<b>qt→isOnline_async(callback, context)</b>
Checks if the quaternion component is currently reachable, without raising any error (asynchronous version).
<b>qt→load(msValidity)</b>
Preloads the quaternion component cache with a specified validity duration.
<b>qt→loadCalibrationPoints(rawValues, refValues)</b>
Retrieves error correction data points previously entered using the method calibrateFromPoints.
<b>qt→load_async(msValidity, callback, context)</b>
Preloads the quaternion component cache with a specified validity duration (asynchronous version).
<b>qt→nextQt()</b>
Continues the enumeration of quaternion components started using yFirstQt( ).
<b>qt→registerTimedReportCallback(callback)</b>
Registers the callback function that is invoked on every periodic timed notification.
<b>qt→registerValueCallback(callback)</b>
Registers the callback function that is invoked on every change of advertised value.
<b>qt→set_highestValue(newval)</b>
Changes the recorded maximal value observed.
<b>qt→set_logFrequency(newval)</b>
Changes the datalogger recording frequency for this function.
<b>qt→set_logicalName(newval)</b>

Changes the logical name of the quaternion component.

**qt→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**qt→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**qt→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**qt→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**qt→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YQt.FindQt() yFindQt()YQt.FindQt()

YQt

Retrieves a quaternion component for a given identifier.

```
def FindQt( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the quaternion component is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YQt.isOnline()` to test if the quaternion component is indeed online at a given time. In case of ambiguity when looking for a quaternion component by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the quaternion component

**Returns :**

a `YQt` object allowing you to drive the quaternion component.

**YQt.FirstQt()****YQt****yFirstQt()YQt.FirstQt()**

Starts the enumeration of quaternion components currently accessible.

```
def FirstQt( )
```

Use the method `YQt.nextQt()` to iterate on next quaternion components.

**Returns :**

a pointer to a `YQt` object, corresponding to the first quaternion component currently online, or a `null` pointer if there are none.

**qt→calibrateFromPoints()|qt.calibrateFromPoints()****YQt**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**qt→describe()qt.describe()****YQt**

Returns a short text that describes unambiguously the instance of the quaternion component in the form TYPE ( NAME )=SERIAL . FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

```
a string that describes the quaternion component (ex:  
Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)
```

---

<b>qt→get_advertisedValue()</b>	<b>YQt</b>
<b>qt→advertisedValue()qt.get_advertisedValue()</b>	

---

Returns the current value of the quaternion component (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the quaternion component (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**qt→get\_currentRawValue()****YQt****qt→currentRawValue()qt.get\_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**qt→get\_currentValue()  
qt→currentValue()qt.get\_currentValue()****YQt**

Returns the current value of the value.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the value

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

**qt→get\_errorMessage()****YQt****qt→errorMessage()qt.get\_errorMessage()**

Returns the error message of the latest error with the quaternion component.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the quaternion component object

**qt→get\_errorType()****YQt****qt→errorType()qt.get\_errorType()**

Returns the numerical error code of the latest error with the quaternion component.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the quaternion component object

**qt→get\_friendlyName()****YQt****qt→friendlyName()qt.get\_friendlyName()**

Returns a global identifier of the quaternion component in the format MODULE\_NAME.FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the quaternion component if they are defined, otherwise the serial number of the module and the hardware identifier of the quaternion component (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the quaternion component using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

---

**qt→get\_functionDescriptor()** YQt  
**qt→functionDescriptor()qt.get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

---

**qt->get\_functionId()****YQt****qt->functionId()qt.get\_functionId()**

Returns the hardware identifier of the quaternion component, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the quaternion component (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**qt→get\_hardwareId()****YQt****qt→hardwareId()qt.get\_hardwareId()**

Returns the unique hardware identifier of the quaternion component in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the quaternion component. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the quaternion component (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**qt→get\_highestValue()****YQt****qt→highestValue()qt.get\_highestValue()**

Returns the maximal value observed for the value since the device was started.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the value since the device was started

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**qt→get\_logFrequency()  
qt→logFrequency()qt.get\_logFrequency()****YQt**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

---

**qt→get\_logicalName()****YQt****qt→logicalName()qt.get\_logicalName()**

Returns the logical name of the quaternion component.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the quaternion component. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**qt→get\_lowestValue()**  
**qt→lowestValue()qt.get\_lowestValue()****YQt**

Returns the minimal value observed for the value since the device was started.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the value since the device was started

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**qt→get\_module()****YQt****qt→module()qt.get\_module()**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**qt→get\_recordedData()  
qt→recordedData()qt.get\_recordedData()****YQt**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime )
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**qt→get\_reportFrequency()****YQt****qt→reportFrequency()qt.get\_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**qt→get\_resolution()****YQt****qt→resolution()qt.get\_resolution()**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

---

**qt→get\_unit()**  
**qt→unit()qt.get\_unit()****YQt**

Returns the measuring unit for the value.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the value

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**qt→get(userData)**  
**qt→userData()qt.get(userData)**

YQt

Returns the value of the userData attribute, as previously stored using method set(userData).

```
def get(userData) 
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**qt→isOnline()qt.isOnline()****YQt**

Checks if the quaternion component is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the quaternion component in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the quaternion component.

**Returns :**

`true` if the quaternion component can be reached, and `false` otherwise

**qt→load()qt.load()****YQt**

Preloads the quaternion component cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**qt→loadCalibrationPoints()qt.loadCalibrationPoints()****YQt**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**qt→nextQt()qt.nextQt()****YQt**

Continues the enumeration of quaternion components started using `yFirstQt()`.

```
def nextQt( )
```

**Returns :**

a pointer to a `YQt` object, corresponding to a quaternion component currently online, or a `null` pointer if there are no more quaternion components to enumerate.

**qt→registerTimedReportCallback()  
qt.registerTimedReportCallback()**

YQt

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**qt→registerValueCallback()  
qt.registerValueCallback()****YQt**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

---

**qt→set\_highestValue()**  
**qt→setHighestValue()qt.set\_highestValue()****YQt**

Changes the recorded maximal value observed.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**qt→set\_logFrequency()** YQt  
**qt→setLogFrequency()qt.set\_logFrequency()**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**qt→set\_logicalName()****YQt****qt→setLogicalName()qt.set\_logicalName()**

Changes the logical name of the quaternion component.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the quaternion component.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**qt→set\_lowestValue()  
qt→setLowestValue()qt.set\_lowestValue()****YQt**

Changes the recorded minimal value observed.

```
def set_lowestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**qt→set\_reportFrequency()**

YQt

**qt→setReportFrequency()qt.set\_reportFrequency()**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**qt→set\_resolution()  
qt→setResolution()qt.set\_resolution()****YQt**

Changes the resolution of the measured physical values.

```
def set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**qt→set(userData)**  
**qt→setUserData()qt.set(userData)****YQt**

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData) data
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.33. Real Time Clock function interface

The RealTimeClock function maintains and provides current date and time, even across power cut lasting several days. It is the base for automated wake-up functions provided by the WakeUpScheduler. The current time may represent a local time as well as an UTC time, but no automatic time change will occur to account for daylight saving time.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_realtimeclock.js'></script>
nodejs var yoctolib = require('yoctolib');
var YRealTimeClock = yoctolib.YRealTimeClock;
php require_once('yocto_realtimeclock.php');
cpp #include "yocto_realtimeclock.h"
m #import "yocto_realtimeclock.h"
pas uses yocto_realtimeclock;
vb yocto_realtimeclock.vb
cs yocto_realtimeclock.cs
java import com.yoctopuce.YoctoAPI.YRealTimeClock;
py from yocto_realtimeclock import *

```

### Global functions

#### **yFindRealTimeClock(func)**

Retrieves a clock for a given identifier.

#### **yFirstRealTimeClock()**

Starts the enumeration of clocks currently accessible.

### YRealTimeClock methods

#### **realtimeclock→describe()**

Returns a short text that describes unambiguously the instance of the clock in the form  
TYPE ( NAME ) = SERIAL . FUNCTIONID.

#### **realtimeclock→get\_advertisedValue()**

Returns the current value of the clock (no more than 6 characters).

#### **realtimeclock→get\_dateTime()**

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

#### **realtimeclock→get\_errorMessage()**

Returns the error message of the latest error with the clock.

#### **realtimeclock→get\_errorType()**

Returns the numerical error code of the latest error with the clock.

#### **realtimeclock→get\_friendlyName()**

Returns a global identifier of the clock in the format MODULE\_NAME . FUNCTION\_NAME.

#### **realtimeclock→get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### **realtimeclock→get\_functionId()**

Returns the hardware identifier of the clock, without reference to the module.

#### **realtimeclock→get\_hardwareId()**

Returns the unique hardware identifier of the clock in the form SERIAL . FUNCTIONID.

#### **realtimeclock→get\_logicalName()**

Returns the logical name of the clock.

#### **realtimeclock→get\_module()**

Gets the YModule object for the device on which the function is located.

**realtimeclock→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**realtimeclock→get\_timeSet()**

Returns true if the clock has been set, and false otherwise.

**realtimeclock→get\_unixTime()**

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

**realtimeclock→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set(userData).

**realtimeclock→get\_utcOffset()**

Returns the number of seconds between current time and UTC time (time zone).

**realtimeclock→isOnline()**

Checks if the clock is currently reachable, without raising any error.

**realtimeclock→isOnline\_async(callback, context)**

Checks if the clock is currently reachable, without raising any error (asynchronous version).

**realtimeclock→load(msValidity)**

Preloads the clock cache with a specified validity duration.

**realtimeclock→load\_async(msValidity, callback, context)**

Preloads the clock cache with a specified validity duration (asynchronous version).

**realtimeclock→nextRealTimeClock()**

Continues the enumeration of clocks started using yFirstRealTimeClock( ).

**realtimeclock→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**realtimeclock→set\_logicalName(newval)**

Changes the logical name of the clock.

**realtimeclock→set\_unixTime(newval)**

Changes the current time.

**realtimeclock→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**realtimeclock→set\_utcOffset(newval)**

Changes the number of seconds between current time and UTC time (time zone).

**realtimeclock→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YRealTimeClock.FindRealTimeClock() yFindRealTimeClock() YRealTimeClock.FindRealTimeClock()

YRealTimeClock

Retrieves a clock for a given identifier.

```
def FindRealTimeClock( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the clock is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRealTimeClock.isOnline()` to test if the clock is indeed online at a given time. In case of ambiguity when looking for a clock by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the clock

### Returns :

a `YRealTimeClock` object allowing you to drive the clock.

**YRealTimeClock.FirstRealTimeClock()****YRealTimeClock****yFirstRealTimeClock()****YRealTimeClock.FirstRealTimeClock()**

---

Starts the enumeration of clocks currently accessible.

```
def FirstRealTimeClock( )
```

Use the method `YRealTimeClock.nextRealTimeClock()` to iterate on next clocks.

**Returns :**

a pointer to a `YRealTimeClock` object, corresponding to the first clock currently online, or a null pointer if there are none.

**realtimeclock→describe()realtimeclock.describe()****YRealTimeClock**

Returns a short text that describes unambiguously the instance of the clock in the form  
TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the clock (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

**realtimeclock→get\_advertisedValue()**  
**realtimeclock→advertisedValue()**  
**realtimeclock.get\_advertisedValue()**

**YRealTimeClock**

Returns the current value of the clock (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the clock (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**realtimeclock→getDateTime()**  
**realtimeclock→dateTime()**  
**realtimeclock.getDateTime()**

---

**YRealTimeClock**

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

```
def getDateTime( )
```

**Returns :**

a string corresponding to the current time in the form "YYYY/MM/DD hh:mm:ss"

On failure, throws an exception or returns Y\_DATETIME\_INVALID.

---

**realtimeclock→get\_errorMessage()**  
**realtimeclock→errorMessage()**  
**realtimeclock.get\_errorMessage()**

**YRealTimeClock**

Returns the error message of the latest error with the clock.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the clock object

**realtimeclock→get\_errorType()**  
**realtimeclock→errorType()**  
**realtimeclock.get\_errorType()**

**YRealTimeClock**

Returns the numerical error code of the latest error with the clock.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the clock object

**realtimeclock→get\_friendlyName()**  
**realtimeclock→friendlyName()**  
**realtimeclock.get\_friendlyName()**

**YRealTimeClock**

Returns a global identifier of the clock in the format MODULE\_NAME . FUNCTION\_NAME.

**def get\_friendlyName( )**

The returned string uses the logical names of the module and of the clock if they are defined, otherwise the serial number of the module and the hardware identifier of the clock (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the clock using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

`realtimeclock→get_functionDescriptor()`  
`realtimeclock→functionDescriptor()`  
`realtimeclock.get_functionDescriptor()`

**YRealTimeClock**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

**Returns :**

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

**realtimeclock→get\_functionId()**  
**realtimeclock→functionId()**  
**realtimeclock.get\_functionId()**

**YRealTimeClock**

Returns the hardware identifier of the clock, without reference to the module.

```
def get_functionId( )
```

For example relay1

**Returns :**

a string that identifies the clock (ex: relay1) On failure, throws an exception or returns Y\_FUNCTIONID\_INVALID.

**realtimeclock→get\_hardwareId()**  
**realtimeclock→hardwareId()**  
**realtimeclock.get\_hardwareId()**

**YRealTimeClock**

Returns the unique hardware identifier of the clock in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the clock. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the clock (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**realtimeclock→get\_logicalName()**  
**realtimeclock→logicalName()**  
**realtimeclock.get\_logicalName()**

**YRealTimeClock**

Returns the logical name of the clock.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the clock. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

---

**realtimeclock→get\_module()** **YRealTimeClock**  
**realtimeclock→module()realtimeclock.get\_module()**

---

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

---

**realtimeclock→get\_timeSet()****YRealTimeClock****realtimeclock→timeSet()realtimeclock.get\_timeSet()**

---

Returns true if the clock has been set, and false otherwise.

```
def get_timeSet( )
```

**Returns :**

either Y\_TIMESSET\_FALSE or Y\_TIMESSET\_TRUE, according to true if the clock has been set, and false otherwise

On failure, throws an exception or returns Y\_TIMESSET\_INVALID.

**realtimeclock→get\_unixTime()**  
**realtimeclock→unixTime()**  
**realtimeclock.get\_unixTime()**

**YRealTimeClock**

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

```
def get_unixTime( )
```

**Returns :**

an integer corresponding to the current time in Unix format (number of elapsed seconds since Jan 1st, 1970)

On failure, throws an exception or returns Y\_UNIXTIME\_INVALID.

---

**realtimeclock→get(userData)**  
**realtimeclock→userData()**  
**realtimeclock.get(userData)**

---

**YRealTimeClock**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

`def get(userData) :`

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**realtimeclock→get\_utcOffset()**  
**realtimeclock→utcOffset()**  
**realtimeclock.get\_utcOffset()**

**YRealTimeClock**

---

Returns the number of seconds between current time and UTC time (time zone).

```
def get_utcOffset( )
```

**Returns :**

an integer corresponding to the number of seconds between current time and UTC time (time zone)

On failure, throws an exception or returns Y\_UTCOFFSET\_INVALID.

**realtimeclock→isOnline()****YRealTimeClock**

Checks if the clock is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the clock in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the clock.

**Returns :**

`true` if the clock can be reached, and `false` otherwise

**realtimeclock→load()|realtimeclock.load()****YRealTimeClock**

Preloads the clock cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**realtimeclock→nextRealTimeClock()**  
**realtimeclock.nextRealTimeClock()****YRealTimeClock**

Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

```
def nextRealTimeClock( )
```

**Returns :**

a pointer to a `YRealTimeClock` object, corresponding to a clock currently online, or a null pointer if there are no more clocks to enumerate.

**realtimeclock→registerValueCallback()  
realtimeclock.registerValueCallback()****YRealTimeClock**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**realtimeclock→set\_logicalName()**  
**realtimeclock→setLogicalName()**  
**realtimeclock.set\_logicalName()**

**YRealTimeClock**

Changes the logical name of the clock.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the clock.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**realtimeclock→set\_unixTime()**  
**realtimeclock→setUnixTime()**  
**realtimeclock.set\_unixTime()**

**YRealTimeClock**

Changes the current time.

```
def set_unixTime( newval )
```

Time is specified in Unix format (number of elapsed seconds since Jan 1st, 1970). If current UTC time is known, utcOffset will be automatically adjusted for the new specified time.

**Parameters :**

**newval** an integer corresponding to the current time

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**realtimeclock→set(userData)**  
**realtimeclock→setUserData()**  
**realtimeclock.set(userData)**

**YRealTimeClock**

Stores a user context provided as argument in the userData attribute of the function.

```
def setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

**realtimeclock→set\_utcOffset()**  
**realtimeclock→setUtcOffset()**  
**realtimeclock.set\_utcOffset()**

**YRealTimeClock**

Changes the number of seconds between current time and UTC time (time zone).

```
def set_utcOffset( newval )
```

The timezone is automatically rounded to the nearest multiple of 15 minutes. If current UTC time is known, the current time will automatically be updated according to the selected time zone.

**Parameters :**

**newval** an integer corresponding to the number of seconds between current time and UTC time (time zone)

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## 3.34. Reference frame configuration

This class is used to setup the base orientation of the Yocto-3D, so that the orientation functions, relative to the earth surface plane, use the proper reference frame. The class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_refframe.js'></script>
node.js	var yoctolib = require('yoctolib');
	var YRefFrame = yoctolib.YRefFrame;
php	require_once('yocto_refframe.php');
cpp	#include "yocto_refframe.h"
m	#import "yocto_refframe.h"
pas	uses yocto_refframe;
vb	yocto_refframe.vb
cs	yocto_refframe.cs
java	import com.yoctopuce.YoctoAPI.YRefFrame;
py	from yocto_refframe import *

### Global functions

#### yFindRefFrame(func)

Retrieves a reference frame for a given identifier.

#### yFirstRefFrame()

Starts the enumeration of reference frames currently accessible.

### YRefFrame methods

#### refframe→cancel3DCalibration()

Aborts the sensors tridimensional calibration process et restores normal settings.

#### refframe→describe()

Returns a short text that describes unambiguously the instance of the reference frame in the form TYPE (NAME) = SERIAL.FUNCTIONID.

#### refframe→get\_3DCalibrationHint()

Returns instructions to proceed to the tridimensional calibration initiated with method start3DCalibration.

#### refframe→get\_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.

#### refframe→get\_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method start3DCalibration.

#### refframe→get\_3DCalibrationStage()

Returns index of the current stage of the calibration initiated with method start3DCalibration.

#### refframe→get\_3DCalibrationStageProgress()

Returns the process indicator for the current stage of the calibration initiated with method start3DCalibration.

#### refframe→get\_advertisedValue()

Returns the current value of the reference frame (no more than 6 characters).

#### refframe→get\_bearing()

Returns the reference bearing used by the compass.

### 3. Reference

<b>reframe→get_errorMessage()</b>	Returns the error message of the latest error with the reference frame.
<b>reframe→get_errorType()</b>	Returns the numerical error code of the latest error with the reference frame.
<b>reframe→get_friendlyName()</b>	Returns a global identifier of the reference frame in the format MODULE_NAME . FUNCTION_NAME.
<b>reframe→get_functionDescriptor()</b>	Returns a unique identifier of type YFUN_DESCR corresponding to the function.
<b>reframe→get_functionId()</b>	Returns the hardware identifier of the reference frame, without reference to the module.
<b>reframe→get_hardwareId()</b>	Returns the unique hardware identifier of the reference frame in the form SERIAL . FUNCTIONID.
<b>reframe→get_logicalName()</b>	Returns the logical name of the reference frame.
<b>reframe→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>reframe→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>reframe→get_mountOrientation()</b>	Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.
<b>reframe→get_mountPosition()</b>	Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.
<b>reframe→get(userData)</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>reframe→isOnline()</b>	Checks if the reference frame is currently reachable, without raising any error.
<b>reframe→isOnline_async(callback, context)</b>	Checks if the reference frame is currently reachable, without raising any error (asynchronous version).
<b>reframe→load(msValidity)</b>	Preloads the reference frame cache with a specified validity duration.
<b>reframe→load_async(msValidity, callback, context)</b>	Preloads the reference frame cache with a specified validity duration (asynchronous version).
<b>reframe→more3DCalibration()</b>	Continues the sensors tridimensional calibration process previously initiated using method start3DCalibration.
<b>reframe→nextRefFrame()</b>	Continues the enumeration of reference frames started using yFirstRefFrame( ).
<b>reframe→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>reframe→save3DCalibration()</b>	Applies the sensors tridimensional calibration parameters that have just been computed.
<b>reframe→set_bearing(newval)</b>	Changes the reference bearing used by the compass.
<b>reframe→set_logicalName(newval)</b>	

Changes the logical name of the reference frame.

**refframe→set\_mountPosition(position, orientation)**

Changes the compass and tilt sensor frame of reference.

**refframe→set(userData)**

Stores a user context provided as argument in the userData attribute of the function.

**refframe→start3DCalibration()**

Initiates the sensors tridimensional calibration process.

**refframe→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YRefFrame.FindRefFrame() yFindRefFrame()YRefFrame.FindRefFrame()

YRefFrame

Retrieves a reference frame for a given identifier.

```
def FindRefFrame( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the reference frame is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRefFrame.isOnline()` to test if the reference frame is indeed online at a given time. In case of ambiguity when looking for a reference frame by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the reference frame

### Returns :

a `YRefFrame` object allowing you to drive the reference frame.

**YRefFrame.FirstRefFrame()****YRefFrame****yFirstRefFrame()YRefFrame.FirstRefFrame()**

Starts the enumeration of reference frames currently accessible.

```
def FirstRefFrame( )
```

Use the method `YRefFrame.nextRefFrame( )` to iterate on next reference frames.

**Returns :**

a pointer to a `YRefFrame` object, corresponding to the first reference frame currently online, or a `null` pointer if there are none.

**refframe→cancel3DCalibration()**  
**refframe.cancel3DCalibration()**

---

**YRefFrame**

Aborts the sensors tridimensional calibration process et restores normal settings.

```
def cancel3DCalibration( )
```

On failure, throws an exception or returns a negative error code.

**refframe→describe()****YRefFrame**

Returns a short text that describes unambiguously the instance of the reference frame in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the reference frame (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**refframe→get\_3DCalibrationHint()**  
**refframe→3DCalibrationHint()**  
**refframe.get\_3DCalibrationHint()**

**YRefFrame**

Returns instructions to proceed to the tridimensional calibration initiated with method start3DCalibration.

```
def get_3DCalibrationHint( )
```

**Returns :**

a character string.

**refframe→get\_3DCalibrationLogMsg()**

**YRefFrame**

**refframe→3DCalibrationLogMsg()**

**refframe.get\_3DCalibrationLogMsg()**

---

Returns the latest log message from the calibration process.

```
def get_3DCalibrationLogMsg( )
```

When no new message is available, returns an empty string.

**Returns :**

a character string.

**refframe→get\_3DCalibrationProgress()**  
**refframe→3DCalibrationProgress()**  
**refframe.get\_3DCalibrationProgress()**

**YRefFrame**

Returns the global process indicator for the tridimensional calibration initiated with method start3DCalibration.

```
def get_3DCalibrationProgress( )
```

**Returns :**

an integer between 0 (not started) and 100 (stage completed).

**refframe→get\_3DCalibrationStage()**  
**refframe→3DCalibrationStage()**  
**refframe.get\_3DCalibrationStage()**

**YRefFrame**

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

```
def get_3DCalibrationStage( )
```

**Returns :**

an integer, growing each time a calibration stage is completed.

`refframe→get_3DCalibrationStageProgress()`

**YRefFrame**

`refframe→3DCalibrationStageProgress()`

`refframe.get_3DCalibrationStageProgress()`

---

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

```
def get_3DCalibrationStageProgress( )
```

**Returns :**

an integer between 0 (not started) and 100 (stage completed).

**refframe→get\_advertisedValue()**  
**refframe→advertisedValue()**  
**refframe.get\_advertisedValue()**

**YRefFrame**

Returns the current value of the reference frame (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the reference frame (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

`refframe→get_bearing()`

**YRefFrame**

`refframe→bearing()refframe.get_bearing()`

---

Returns the reference bearing used by the compass.

```
def get_bearing( )
```

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here.

**Returns :**

a floating point number corresponding to the reference bearing used by the compass

On failure, throws an exception or returns Y\_BEARING\_INVALID.

**refframe→getErrorMessage()**  
**refframe→errorMessage()**  
**refframe.getErrorMessage()**

**YRefFrame**

Returns the error message of the latest error with the reference frame.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the reference frame object

**refframe→get\_errorType()**

**YRefFrame**

**refframe→errorType()refframe.get\_errorType()**

---

Returns the numerical error code of the latest error with the reference frame.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the reference frame object

**refframe→get\_friendlyName()****YRefFrame****refframe→friendlyName()refframe.get\_friendlyName()**

Returns a global identifier of the reference frame in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the reference frame if they are defined, otherwise the serial number of the module and the hardware identifier of the reference frame (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the reference frame using logical names (ex: MyCustomName . relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

`refframe→get_functionDescriptor()`  
`refframe→functionDescriptor()`  
`refframe.get_functionDescriptor()`

**YRefFrame**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

---

**refframe→get\_functionId()****YRefFrame****refframe→functionId()refframe.get\_functionId()**

---

Returns the hardware identifier of the reference frame, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the reference frame (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**refframe→get\_hwId()**

**YRefFrame**

**refframe→hwId()refframe.get\_hwId()**

---

Returns the unique hardware identifier of the reference frame in the form SERIAL.FUNCTIONID.

```
def get_hwId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the reference frame. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the reference frame (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

---

**refframe→get\_logicalName()****YRefFrame****refframe→logicalName()refframe.get\_logicalName()**

---

Returns the logical name of the reference frame.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the reference frame. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**refframe→get\_module()****YRefFrame****refframe→module()refframe.get\_module()**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

**refframe→get\_mountOrientation()**  
**refframe→mountOrientation()**  
**refframe.get\_mountOrientation()**

**YRefFrame**

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

```
def get_mountOrientation( )
```

**Returns :**

a value among the enumeration Y\_MOUNTORIENTATION (Y\_MOUNTORIENTATION\_TWELVE, Y\_MOUNTORIENTATION\_THREE, Y\_MOUNTORIENTATION\_SIX, Y\_MOUNTORIENTATION\_NINE) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear.

On failure, throws an exception or returns a negative error code.

**refframe→get\_mountPosition()**  
**refframe→mountPosition()**  
**refframe.get\_mountPosition()**

**YRefFrame**

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

```
def get_mountPosition( )
```

**Returns :**

a value among the Y\_MOUNTPOSITION enumeration (Y\_MOUNTPOSITION\_BOTTOM, Y\_MOUNTPOSITION\_TOP, Y\_MOUNTPOSITION\_FRONT, Y\_MOUNTPOSITION\_RIGHT, Y\_MOUNTPOSITION\_REAR, Y\_MOUNTPOSITION\_LEFT), corresponding to the installation in a box, on one of the six faces.

On failure, throws an exception or returns a negative error code.

---

**refframe→get(userData)****YRefFrame****refframe→userData()refframe.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**refframe→isOnline()refframe.isOnline()****YRefFrame**

Checks if the reference frame is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the reference frame in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the reference frame.

**Returns :**

true if the reference frame can be reached, and false otherwise

**refframe→load()refframe.load()****YRefFrame**

Preloads the reference frame cache with a specified validity duration.

```
def load( msValidity )
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**refframe→more3DCalibration()  
refframe.more3DCalibration()****YRefFrame**

Continues the sensors tridimensional calibration process previously initiated using method start3DCalibration.

```
def more3DCalibration( )
```

This method should be called approximately 5 times per second, while positioning the device according to the instructions provided by method get\_3DCalibrationHint. Note that the instructions change during the calibration process. On failure, throws an exception or returns a negative error code.

**refframe→nextRefFrame()****YRefFrame**

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

```
def nextRefFrame( )
```

**Returns :**

a pointer to a `YRefFrame` object, corresponding to a reference frame currently online, or a null pointer if there are no more reference frames to enumerate.

**refframe→registerValueCallback()**  
**refframe.registerValueCallback()****YRefFrame**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

---

**refframe→save3DCalibration()**  
**refframe.save3DCalibration()****YRefFrame**

Applies the sensors tridimensional calibration parameters that have just been computed.

```
def save3DCalibration( )
```

Remember to call the `saveToFlash( )` method of the module if the changes must be kept when the device is restarted. On failure, throws an exception or returns a negative error code.

**refframe→set\_bearing()** YRefFrame  
**refframe→setBearing()refframe.set\_bearing()**

---

Changes the reference bearing used by the compass.

```
def set_bearing( newval)
```

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here. For instance, if you setup as reference bearing the value of the earth magnetic declination, the compass will provide the orientation relative to the geographic North. Similarly, when the sensor is not mounted along the standard directions because it has an additional yaw angle, you can set this angle in the reference bearing so that the compass provides the expected natural direction. Remember to call the saveToFlash( ) method of the module if the modification must be kept.

**Parameters :**

**newval** a floating point number corresponding to the reference bearing used by the compass

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**refframe→set\_logicalName()**  
**refframe→setLogicalName()**  
**refframe.set\_logicalName()**

**YRefFrame**

Changes the logical name of the reference frame.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the reference frame.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

`refframe→set_mountPosition()`  
`refframe→setMountPosition()`  
`refframe.set_mountPosition()`

**YRefFrame**

Changes the compass and tilt sensor frame of reference.

```
def set_mountPosition( position, orientation)
```

The magnetic compass and the tilt sensors (pitch and roll) naturally work in the plane parallel to the earth surface. In case the device is not installed upright and horizontally, you must select its reference orientation (parallel to the earth surface) so that the measures are made relative to this position.

**Parameters :**

**position** a value among the `Y_MOUNTPOSITION` enumeration (`Y_MOUNTPOSITION_BOTTOM`,  
`Y_MOUNTPOSITION_TOP`, `Y_MOUNTPOSITION_FRONT`,  
`Y_MOUNTPOSITION_RIGHT`, `Y_MOUNTPOSITION_REAR`,  
`Y_MOUNTPOSITION_LEFT`), corresponding to the installation in a box, on one of the six faces.

**orientation** a value among the enumeration `Y_MOUNTORIENTATION` (`Y_MOUNTORIENTATION_TWELVE`, `Y_MOUNTORIENTATION_THREE`,  
`Y_MOUNTORIENTATION_SIX`, `Y_MOUNTORIENTATION_NINE`) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

---

**refframe→set(userData)****YRefFrame****refframe→setUserData()refframe.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData) data
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

**refframe→start3DCalibration()**  
**refframe.start3DCalibration()****YRefFrame**

Initiates the sensors tridimensional calibration process.

```
def start3DCalibration( )
```

This calibration is used at low level for inertial position estimation and to enhance the precision of the tilt sensors. After calling this method, the device should be moved according to the instructions provided by method `get_3DCalibrationHint`, and `more3DCalibration` should be invoked about 5 times per second. The calibration procedure is completed when the method `get_3DCalibrationProgress` returns 100. At this point, the computed calibration parameters can be applied using method `save3DCalibration`. The calibration process can be canceled at any time using method `cancel3DCalibration`. On failure, throws an exception or returns a negative error code.

## 3.35. Relay function interface

The Yoctopuce application programming interface allows you to switch the relay state. This change is not persistent: the relay will automatically return to its idle position whenever power is lost or if the module is restarted. The library can also generate automatically short pulses of determined duration. On devices with two output for each relay (double throw), the two outputs are named A and B, with output A corresponding to the idle position (at power off) and the output B corresponding to the active state. If you prefer the alternate default state, simply switch your cables on the board.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_relay.js'></script>
node.js	var yoctolib = require('yoctolib');
php	var YRelay = yoctolib.YRelay;
require_once('yocto_relay.php');	
cpp	#include "yocto_relay.h"
m	#import "yocto_relay.h"
pas	uses yocto_relay;
vb	yocto_relay.vb
cs	yocto_relay.cs
java	import com.yoctopuce.YoctoAPI.YRelay;
py	from yocto_relay import *

### Global functions

#### yFindRelay(func)

Retrieves a relay for a given identifier.

#### yFirstRelay()

Starts the enumeration of relays currently accessible.

### YRelay methods

#### relay->delayedPulse(ms\_delay, ms\_duration)

Schedules a pulse.

#### relay->describe()

Returns a short text that describes unambiguously the instance of the relay in the form TYPE(NAME)=SERIAL.FUNCTIONID.

#### relay->get\_advertisedValue()

Returns the current value of the relay (no more than 6 characters).

#### relay->get\_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call). When there is no scheduled pulse, returns zero.

#### relay->get\_errorMessage()

Returns the error message of the latest error with the relay.

#### relay->get\_errorType()

Returns the numerical error code of the latest error with the relay.

#### relay->get\_friendlyName()

Returns a global identifier of the relay in the format MODULE\_NAME . FUNCTION\_NAME.

#### relay->get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### relay->get\_functionId()

Returns the hardware identifier of the relay, without reference to the module.

#### relay->get\_hardwareId()

	Returns the unique hardware identifier of the relay in the form SERIAL.FUNCTIONID.
<b>relay→get_logicalName()</b>	Returns the logical name of the relay.
<b>relay→get_maxTimeOnStateA()</b>	Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.
<b>relay→get_maxTimeOnStateB()</b>	Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.
<b>relay→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>relay→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>relay→get_output()</b>	Returns the output state of the relays, when used as a simple switch (single throw).
<b>relay→get_pulseTimer()</b>	Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.
<b>relay→get_state()</b>	Returns the state of the relays (A for the idle position, B for the active position).
<b>relay→get_stateAtPowerOn()</b>	Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).
<b>relay→get(userData)</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>relay→isOnline()</b>	Checks if the relay is currently reachable, without raising any error.
<b>relay→isOnline_async(callback, context)</b>	Checks if the relay is currently reachable, without raising any error (asynchronous version).
<b>relay→load(msValidity)</b>	Preloads the relay cache with a specified validity duration.
<b>relay→load_async(msValidity, callback, context)</b>	Preloads the relay cache with a specified validity duration (asynchronous version).
<b>relay→nextRelay()</b>	Continues the enumeration of relays started using yFirstRelay( ).
<b>relay→pulse(ms_duration)</b>	Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).
<b>relay→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>relay→set_logicalName(newval)</b>	Changes the logical name of the relay.
<b>relay→set_maxTimeOnStateA(newval)</b>	Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.
<b>relay→set_maxTimeOnStateB(newval)</b>	

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

**relay→set\_output(newval)**

Changes the output state of the relays, when used as a simple switch (single throw).

**relay→set\_state(newval)**

Changes the state of the relays (A for the idle position, B for the active position).

**relay→set\_stateAtPowerOn(newval)**

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

**relay→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**relay→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YRelay.FindRelay() yFindRelay()YRelay.FindRelay()

YRelay

Retrieves a relay for a given identifier.

```
def FindRelay( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the relay is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YRelay.isOnline()` to test if the relay is indeed online at a given time. In case of ambiguity when looking for a relay by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the relay

### Returns :

a `YRelay` object allowing you to drive the relay.

**YRelay.FirstRelay()****YRelay****yFirstRelay()YRelay.FirstRelay()**

Starts the enumeration of relays currently accessible.

```
def FirstRelay( )
```

Use the method `YRelay.nextRelay( )` to iterate on next relays.

**Returns :**

a pointer to a `YRelay` object, corresponding to the first relay currently online, or a `null` pointer if there are none.

**relay→delayedPulse()relay.delayedPulse()**

YRelay

Schedules a pulse.

```
def delayedPulse( ms_delay, ms_duration)
```

**Parameters :**

**ms\_delay** waiting time before the pulse, in millisecondes

**ms\_duration** pulse duration, in millisecondes

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## relay→describe()relay.describe()

## YRelay

Returns a short text that describes unambiguously the instance of the relay in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the relay (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**relay→get\_advertisedValue()**

**YRelay**

**relay→advertisedValue()relay.get\_advertisedValue()**

---

Returns the current value of the relay (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the relay (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**relay→get\_countdown()****YRelay****relay→countdown()relay.get\_countdown()**

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

```
def get_countdown( )
```

**Returns :**

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns Y\_COUNTDOWN\_INVALID.

**relay→get\_errorMessage()**

**YRelay**

**relay→errorMessage()relay.getErrorMessage()**

---

Returns the error message of the latest error with the relay.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the relay object

---

**relay→get\_errorType()****YRelay****relay→errorType()relay.get\_errorType()**

---

Returns the numerical error code of the latest error with the relay.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the relay object

---

<b>relay→get_friendlyName()</b>	<b>YRelay</b>
<b>relay→friendlyName()relay.get_friendlyName()</b>	

---

Returns a global identifier of the relay in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the relay if they are defined, otherwise the serial number of the module and the hardware identifier of the relay (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the relay using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**relay→get\_functionDescriptor()**  
**relay→functionDescriptor()**  
**relay.get\_functionDescriptor()**

**YRelay**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

<b>relay-&gt;get_functionId()</b>	<b>YRelay</b>
<b>relay-&gt;functionId()relay.get_functionId()</b>	

---

Returns the hardware identifier of the relay, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the relay (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**relay→get\_hardwareId()****YRelay****relay→hardwareId()relay.get\_hardwareId()**

Returns the unique hardware identifier of the relay in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the relay. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the relay (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**relay→get\_logicalName()** YRelay  
**relay→logicalName()relay.get\_logicalName()**

---

Returns the logical name of the relay.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the relay. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**relay→get\_maxTimeOnStateA()**  
**relay→maxTimeOnStateA()**  
**relay.get\_maxTimeOnStateA()**

**YRelay**

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

```
def get_maxTimeOnStateA( )
```

Zero means no maximum time.

**Returns :**

an integer

On failure, throws an exception or returns Y\_MAXTIMEONSTATEA\_INVALID.

**relay→get\_maxTimeOnStateB()**  
**relay→maxTimeOnStateB()**  
**relay.get\_maxTimeOnStateB()**

YRelay

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

```
def get_maxTimeOnStateB( )
```

Zero means no maximum time.

**Returns :**

an integer

On failure, throws an exception or returns Y\_MAXTIMEONSTATEB\_INVALID.

**relay→get\_module()****YRelay****relay→module()relay.get\_module()**

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

**relay->get\_output()**

**YRelay**

**relay->output()>relay.get\_output()**

---

Returns the output state of the relays, when used as a simple switch (single throw).

```
def get_output( )
```

**Returns :**

either Y\_OUTPUT\_OFF or Y\_OUTPUT\_ON, according to the output state of the relays, when used as a simple switch (single throw)

On failure, throws an exception or returns Y\_OUTPUT\_INVALID.

**relay→get\_pulseTimer()****YRelay****relay→pulseTimer()relay.get\_pulseTimer()**

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

```
def get_pulseTimer( )
```

When there is no ongoing pulse, returns zero.

**Returns :**

an integer corresponding to the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns Y\_PULSE\_TIMER\_INVALID.

**relay->get\_state()**

**YRelay**

**relay->state()relay.get\_state()**

---

Returns the state of the relays (A for the idle position, B for the active position).

```
def get_state( )
```

**Returns :**

either Y\_STATE\_A or Y\_STATE\_B, according to the state of the relays (A for the idle position, B for the active position)

On failure, throws an exception or returns Y\_STATE\_INVALID.

**relay→get\_stateAtPowerOn()****YRelay****relay→stateAtPowerOn()relay.get\_stateAtPowerOn()**

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

```
def get_stateAtPowerOn( )
```

**Returns :**

a value among Y\_STATEATPOWERON\_UNCHANGED, Y\_STATEATPOWERON\_A and Y\_STATEATPOWERON\_B corresponding to the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns Y\_STATEATPOWERON\_INVALID.

**relay→get(userData)**

**YRelay**

**relay→userData()relay.get(userData)**

---

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**relay→isOnline()relay.isOnline()****YRelay**

Checks if the relay is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the relay in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the relay.

**Returns :**

`true` if the relay can be reached, and `false` otherwise

**relay→load()relay.load()****YRelay**

Preloads the relay cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**relay→nextRelay()**relay.nextRelay()******YRelay**

Continues the enumeration of relays started using `yFirstRelay()`.

```
def nextRelay( )
```

**Returns :**

a pointer to a `YRelay` object, corresponding to a relay currently online, or a `null` pointer if there are no more relays to enumerate.

**relay→pulse()**relay.pulse()******YRelay**

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

```
def pulse( ms_duration )
```

**Parameters :**

**ms\_duration** pulse duration, in millisecondes

**Returns :**

**YAPI\_SUCCESS** if the call succeeds.

On failure, throws an exception or returns a negative error code.

**relay→registerValueCallback()  
relay.registerValueCallback()****YRelay**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

---

<b>relay→set_logicalName()</b>	<b>YRelay</b>
<b>relay→setLogicalName()relay.set_logicalName()</b>	

---

Changes the logical name of the relay.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the relay.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**relay→set\_maxTimeOnStateA()**  
**relay→setMaxTimeOnStateA()**  
**relay.set\_maxTimeOnStateA()**

**YRelay**

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

```
def set_maxTimeOnStateA( newval)
```

Use zero for no maximum time.

**Parameters :**

**newval** an integer

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**relay→set\_maxTimeOnStateB()**  
**relay→setMaxTimeOnStateB()**  
**relay.set\_maxTimeOnStateB()**

YRelay

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

```
def set_maxTimeOnStateB( newval )
```

Use zero for no maximum time.

**Parameters :**

**newval** an integer

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**relay->set\_output()****YRelay****relay->setOutput()relay.set\_output()**

Changes the output state of the relays, when used as a simple switch (single throw).

```
def set_output( newval)
```

**Parameters :**

**newval** either Y\_OUTPUT\_OFF or Y\_OUTPUT\_ON, according to the output state of the relays, when used as a simple switch (single throw)

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**relay→set\_state()****YRelay****relay→setState()relay.set\_state()**

Changes the state of the relays (A for the idle position, B for the active position).

```
def set_state( newval )
```

**Parameters :**

**newval** either Y\_STATE\_A or Y\_STATE\_B, according to the state of the relays (A for the idle position, B for the active position)

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**relay→set\_stateAtPowerOn()**  
**relay→setStateAtPowerOn()**  
**relay.set\_stateAtPowerOn()**

**YRelay**

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

```
def set_stateAtPowerOn( newval )
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters :**

**newval** a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**relay→set(userData)**

**YRelay**

**relay→setUserData()relay.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.36. Sensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_api.js'></script>
nodejs var yoctolib = require('yoctolib');
var YAPI = yoctolib.YAPI;
var YModule = yoctolib.YModule;
php require_once('yocto_api.php');
cpp #include "yocto_api.h"
m #import "yocto_api.h"
pas uses yocto_api;
vb yocto_api.vb
cs yocto_api.cs
java import com.yoctopuce.YoctoAPI.YModule;
py from yocto_api import *

```

### Global functions

#### **yFindSensor(func)**

Retrieves a sensor for a given identifier.

#### **yFirstSensor()**

Starts the enumeration of sensors currently accessible.

### YSensor methods

#### **sensor→calibrateFromPoints(rawValues, refValues)**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### **sensor→describe()**

Returns a short text that describes unambiguously the instance of the sensor in the form TYPE (NAME) = SERIAL.FUNCTIONID.

#### **sensor→get\_advertisedValue()**

Returns the current value of the sensor (no more than 6 characters).

#### **sensor→get\_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### **sensor→get\_currentValue()**

Returns the current value of the measure.

#### **sensor→get\_errorMessage()**

Returns the error message of the latest error with the sensor.

#### **sensor→get\_errorType()**

Returns the numerical error code of the latest error with the sensor.

#### **sensor→get\_friendlyName()**

Returns a global identifier of the sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### **sensor→get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### **sensor→get\_functionId()**

Returns the hardware identifier of the sensor, without reference to the module.

#### **sensor→get\_hardwareId()**

### 3. Reference

Returns the unique hardware identifier of the sensor in the form SERIAL.FUNCTIONID.
<b>sensor-&gt;get_highestValue()</b> Returns the maximal value observed for the measure since the device was started.
<b>sensor-&gt;get_logFrequency()</b> Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>sensor-&gt;get_logicalName()</b> Returns the logical name of the sensor.
<b>sensor-&gt;get_lowestValue()</b> Returns the minimal value observed for the measure since the device was started.
<b>sensor-&gt;get_module()</b> Gets the YModule object for the device on which the function is located.
<b>sensor-&gt;get_module_async(callback, context)</b> Gets the YModule object for the device on which the function is located (asynchronous version).
<b>sensor-&gt;get_recordedData(startTime, endTime)</b> Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>sensor-&gt;get_reportFrequency()</b> Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>sensor-&gt;get_resolution()</b> Returns the resolution of the measured values.
<b>sensor-&gt;get_unit()</b> Returns the measuring unit for the measure.
<b>sensor-&gt;get_userData()</b> Returns the value of the userData attribute, as previously stored using method set(userData).
<b>sensor-&gt;isOnline()</b> Checks if the sensor is currently reachable, without raising any error.
<b>sensor-&gt;isOnline_async(callback, context)</b> Checks if the sensor is currently reachable, without raising any error (asynchronous version).
<b>sensor-&gt;load(msValidity)</b> Preloads the sensor cache with a specified validity duration.
<b>sensor-&gt;loadCalibrationPoints(rawValues, refValues)</b> Retrieves error correction data points previously entered using the method calibrateFromPoints.
<b>sensor-&gt;load_async(msValidity, callback, context)</b> Preloads the sensor cache with a specified validity duration (asynchronous version).
<b>sensor-&gt;nextSensor()</b> Continues the enumeration of sensors started using yFirstSensor( ).
<b>sensor-&gt;registerTimedReportCallback(callback)</b> Registers the callback function that is invoked on every periodic timed notification.
<b>sensor-&gt;registerValueCallback(callback)</b> Registers the callback function that is invoked on every change of advertised value.
<b>sensor-&gt;set_highestValue(newval)</b> Changes the recorded maximal value observed.
<b>sensor-&gt;set_logFrequency(newval)</b> Changes the datalogger recording frequency for this function.
<b>sensor-&gt;set_logicalName(newval)</b>

Changes the logical name of the sensor.

**sensor→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**sensor→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**sensor→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**sensor→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**sensor→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YSensor.FindSensor() yFindSensor()YSensor.FindSensor()

YSensor

Retrieves a sensor for a given identifier.

```
def FindSensor( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSensor.isOnline()` to test if the sensor is indeed online at a given time. In case of ambiguity when looking for a sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the sensor

### Returns :

a `YSensor` object allowing you to drive the sensor.

**YSensor.FirstSensor()****YSensor****yFirstSensor()YSensor.FirstSensor()**

Starts the enumeration of sensors currently accessible.

```
def FirstSensor( )
```

Use the method `YSensor.nextSensor()` to iterate on next sensors.

**Returns :**

a pointer to a `YSensor` object, corresponding to the first sensor currently online, or a null pointer if there are none.

**sensor→calibrateFromPoints()  
sensor.calibrateFromPoints()****YSensor**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**sensor→describe()sensor.describe()****YSensor**

Returns a short text that describes unambiguously the instance of the sensor in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the sensor (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**sensor→get\_advertisedValue()**  
**sensor→advertisedValue()**  
**sensor.get\_advertisedValue()**

**YSensor**

---

Returns the current value of the sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

---

**sensor→get\_currentRawValue()**  
**sensor→currentRawValue()**  
**sensor.get\_currentRawValue()**

**YSensor**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**sensor→get\_currentValue()**

**YSensor**

**sensor→currentValue()sensor.get\_currentValue()**

---

Returns the current value of the measure.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the measure

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

---

**sensor→get\_errorMessage()****YSensor****sensor→errorMessage()sensor.get\_errorMessage()**

---

Returns the error message of the latest error with the sensor.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the sensor object

**sensor→get\_errorType()**  
**sensor→errorType()sensor.get\_errorType()**

---

**YSensor**

Returns the numerical error code of the latest error with the sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the sensor object

**sensor→get\_friendlyName()****YSensor****sensor→friendlyName()sensor.get\_friendlyName()**

Returns a global identifier of the sensor in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the sensor (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the sensor using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**sensor→get\_functionDescriptor()**  
**sensor→functionDescriptor()**  
**sensor.get\_functionDescriptor()**

**YSensor**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

---

**sensor→get\_functionId()****YSensor****sensor→functionId()sensor.get\_functionId()**

---

Returns the hardware identifier of the sensor, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

---

<b>sensor→get_hardwareId()</b>	<b>YSensor</b>
<b>sensor→hardwareId()sensor.get_hardwareId()</b>	

---

Returns the unique hardware identifier of the sensor in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**sensor→get\_highestValue()****YSensor****sensor→highestValue()sensor.get\_highestValue()**

Returns the maximal value observed for the measure since the device was started.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**sensor→get\_logFrequency()** YSensor  
**sensor→logFrequency()sensor.get\_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

---

**sensor→get\_logicalName()****YSensor****sensor→logicalName()sensor.get\_logicalName()**

---

Returns the logical name of the sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**sensor→get\_lowestValue()**

**YSensor**

**sensor→lowestValue()sensor.get\_lowestValue()**

---

Returns the minimal value observed for the measure since the device was started.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

---

**sensor→get\_module()****YSensor****sensor→module()sensor.get\_module()**

---

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

---

<b>sensor→get_recordedData()</b>	<b>YSensor</b>
<b>sensor→recordedData()sensor.get_recordedData()</b>	

---

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime )
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**sensor→get\_reportFrequency()**  
**sensor→reportFrequency()**  
**sensor.get\_reportFrequency()**

**YSensor**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**sensor→get\_resolution()**  
**sensor→resolution()sensor.get\_resolution()**

---

**YSensor**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

---

**sensor→get\_unit()****YSensor****sensor→unit()sensor.get\_unit()**

---

Returns the measuring unit for the measure.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**sensor→get(userData)**

**YSensor**

**sensor→userData()sensor.get(userData)**

---

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**sensor→isOnline()sensor.isOnline()****YSensor**

Checks if the sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the sensor.

**Returns :**

`true` if the sensor can be reached, and `false` otherwise

**sensor→load()sensor.load()****YSensor**

Preloads the sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**sensor→loadCalibrationPoints()  
sensor.loadCalibrationPoints()****YSensor**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## **sensor→nextSensor()sensor.nextSensor()**

**YSensor**

---

Continues the enumeration of sensors started using `yFirstSensor()`.

```
def nextSensor( )
```

**Returns :**

a pointer to a `YSensor` object, corresponding to a sensor currently online, or a `null` pointer if there are no more sensors to enumerate.

**sensor→registerTimedReportCallback()**  
**sensor.registerTimedReportCallback()****YSensor**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**sensor→registerValueCallback()  
sensor.registerValueCallback()****YSensor**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

---

<b>sensor→set_highestValue()</b>	<b>YSensor</b>
<b>sensor→setHighestValue()sensor.set_highestValue()</b>	

---

Changes the recorded maximal value observed.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**sensor->set\_logFrequency()**  
**sensor->setLogFrequency()**  
**sensor.set\_logFrequency()**

**YSensor**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**sensor→set\_logicalName()****YSensor****sensor→setLogicalName()sensor.set\_logicalName()**

Changes the logical name of the sensor.

```
def set_logicalName( newval )
```

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

---

<b>sensor-&gt;set_lowestValue()</b>	<b>YSensor</b>
<b>sensor-&gt;setLowestValue()sensor.set_lowestValue()</b>	

---

Changes the recorded minimal value observed.

```
def set_lowestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**sensor→set\_reportFrequency()**  
**sensor→setReportFrequency()**  
**sensor.set\_reportFrequency()**

**YSensor**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

<b>sensor→set_resolution()</b>	<b>YSensor</b>
<b>sensor→setResolution()sensor.set_resolution()</b>	

---

Changes the resolution of the measured physical values.

```
def set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**sensor→set(userData)****YSensor****sensor→setUserData()|sensor.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.37. Servo function interface

Yoctopuce application programming interface allows you not only to move a servo to a given position, but also to specify the time interval in which the move should be performed. This makes it possible to synchronize two servos involved in a same move.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_servo.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YServo = yoctolib.YServo;
php	require_once('yocto_servo.php');
cpp	#include "yocto_servo.h"
m	#import "yocto_servo.h"
pas	uses yocto_servo;
vb	yocto_servo.vb
cs	yocto_servo.cs
java	import com.yoctopuce.YoctoAPI.YServo;
py	from yocto_servo import *

### Global functions

#### yFindServo(func)

Retrieves a servo for a given identifier.

#### yFirstServo()

Starts the enumeration of servos currently accessible.

### YServo methods

#### servo→describe()

Returns a short text that describes unambiguously the instance of the servo in the form TYPE(NAME)=SERIAL.FUNCTIONID.

#### servo→get\_advertisedValue()

Returns the current value of the servo (no more than 6 characters).

#### servo→get\_enabled()

Returns the state of the servos.

#### servo→get\_enabledAtPowerOn()

Returns the servo signal generator state at power up.

#### servo→get\_errorMessage()

Returns the error message of the latest error with the servo.

#### servo→get\_errorType()

Returns the numerical error code of the latest error with the servo.

#### servo→get\_friendlyName()

Returns a global identifier of the servo in the format MODULE\_NAME . FUNCTION\_NAME.

#### servo→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### servo→get\_functionId()

Returns the hardware identifier of the servo, without reference to the module.

#### servo→get\_hardwareId()

Returns the unique hardware identifier of the servo in the form SERIAL.FUNCTIONID.

#### servo→get\_logicalName()

Returns the logical name of the servo.

**`servo→get_module()`**

Gets the YModule object for the device on which the function is located.

**`servo→get_module_async(callback, context)`**

Gets the YModule object for the device on which the function is located (asynchronous version).

**`servo→get_neutral()`**

Returns the duration in microseconds of a neutral pulse for the servo.

**`servo→get_position()`**

Returns the current servo position.

**`servo→get_positionAtPowerOn()`**

Returns the servo position at device power up.

**`servo→get_range()`**

Returns the current range of use of the servo.

**`servo→get_userData()`**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

**`servo→isOnline()`**

Checks if the servo is currently reachable, without raising any error.

**`servo→isOnline_async(callback, context)`**

Checks if the servo is currently reachable, without raising any error (asynchronous version).

**`servo→load(msValidity)`**

Preloads the servo cache with a specified validity duration.

**`servo→load_async(msValidity, callback, context)`**

Preloads the servo cache with a specified validity duration (asynchronous version).

**`servo→move(target, ms_duration)`**

Performs a smooth move at constant speed toward a given position.

**`servo→nextServo()`**

Continues the enumeration of servos started using `yFirstServo()`.

**`servo→registerValueCallback(callback)`**

Registers the callback function that is invoked on every change of advertised value.

**`servo→set_enabled(newval)`**

Stops or starts the servo.

**`servo→set_enabledAtPowerOn(newval)`**

Configure the servo signal generator state at power up.

**`servo→set_logicalName(newval)`**

Changes the logical name of the servo.

**`servo→set_neutral(newval)`**

Changes the duration of the pulse corresponding to the neutral position of the servo.

**`servo→set_position(newval)`**

Changes immediately the servo driving position.

**`servo→set_positionAtPowerOn(newval)`**

Configure the servo position at device power up.

**`servo→set_range(newval)`**

Changes the range of use of the servo, specified in per cents.

**`servo→set_userData(data)`**

Stores a user context provided as argument in the userData attribute of the function.

**`servo→wait_async(callback, context)`**

### **3. Reference**

---

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YServo.FindServo() yFindServo()YServo.FindServo()

YServo

Retrieves a servo for a given identifier.

```
def FindServo( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the servo is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YServo.isOnline()` to test if the servo is indeed online at a given time. In case of ambiguity when looking for a servo by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the servo

### Returns :

a `YServo` object allowing you to drive the servo.

## **YServo.FirstServo()**

**YServo**

## **yFirstServo()YServo.FirstServo()**

---

Starts the enumeration of servos currently accessible.

```
def FirstServo( )
```

Use the method `YServo.nextServo()` to iterate on next servos.

**Returns :**

a pointer to a `YServo` object, corresponding to the first servo currently online, or a `null` pointer if there are none.

**servo→describe()servo.describe()****YServo**

Returns a short text that describes unambiguously the instance of the servo in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the servo (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**servo→get\_advertisedValue()** YServo  
**servo→advertisedValue()servo.get\_advertisedValue()**

---

Returns the current value of the servo (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the servo (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

---

**servo→get\_enabled()****YServo****servo→enabled()servo.get\_enabled()**

---

Returns the state of the servos.

```
def get_enabled( )
```

**Returns :**

either Y\_ENABLED\_FALSE or Y\_ENABLED\_TRUE, according to the state of the servos

On failure, throws an exception or returns Y\_ENABLED\_INVALID.

**servo→get\_enabledAtPowerOn()**  
**servo→enabledAtPowerOn()**  
**servo.get\_enabledAtPowerOn()**

**YServo**

Returns the servo signal generator state at power up.

```
def get_enabledAtPowerOn( )
```

**Returns :**

either Y\_ENABLEDATPOWERON\_FALSE or Y\_ENABLEDATPOWERON\_TRUE, according to the servo signal generator state at power up

On failure, throws an exception or returns Y\_ENABLEDATPOWERON\_INVALID.

---

**servo→get\_errorMessage()**  
**servo→errorMessage()servo.getErrorMessage()****YServo**

Returns the error message of the latest error with the servo.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the servo object

**servo→get\_errorType()**  
**servo→errorType()servo.get\_errorType()**

---

**YServo**

Returns the numerical error code of the latest error with the servo.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the servo object

---

**servo→get\_friendlyName()****YServo****servo→friendlyName()servo.get\_friendlyName()**

---

Returns a global identifier of the servo in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the servo if they are defined, otherwise the serial number of the module and the hardware identifier of the servo (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the servo using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**servo→get\_functionDescriptor()**  
**servo→functionDescriptor()**  
**servo.get\_functionDescriptor()**

**YServo**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

---

**servo→get\_functionId()****YServo****servo→functionId()servo.get\_functionId()**

---

Returns the hardware identifier of the servo, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the servo (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**servo→get\_hardwareId()**

**YServo**

**servo→hardwareId()servo.get\_hardwareId()**

---

Returns the unique hardware identifier of the servo in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the servo. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the servo (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**servo→get\_logicalName()****YServo****servo→logicalName()servo.get\_logicalName()**

Returns the logical name of the servo.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the servo. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**servo→get\_module()****servo→module()servo.get\_module()****YServo**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**servo→get\_neutral()****YServo****servo→neutral()servo.get\_neutral()**

Returns the duration in microseconds of a neutral pulse for the servo.

```
def get_neutral( )
```

**Returns :**

an integer corresponding to the duration in microseconds of a neutral pulse for the servo

On failure, throws an exception or returns Y\_NEUTRAL\_INVALID.

**servo→get\_position()**  
**servo→position()servo.get\_position()**

---

YServo

Returns the current servo position.

```
def get_position( )
```

**Returns :**

an integer corresponding to the current servo position

On failure, throws an exception or returns Y\_POSITION\_INVALID.

**servo→get\_positionAtPowerOn()**  
**servo→positionAtPowerOn()**  
**servo.get\_positionAtPowerOn()**

**YServo**

Returns the servo position at device power up.

```
def get_positionAtPowerOn( )
```

**Returns :**

an integer corresponding to the servo position at device power up

On failure, throws an exception or returns Y\_POSITIONATPOWERON\_INVALID.

**servo→get\_range()**

**YServo**

**servo→range()servo.get\_range()**

---

Returns the current range of use of the servo.

```
def get_range( )
```

**Returns :**

an integer corresponding to the current range of use of the servo

On failure, throws an exception or returns Y\_RANGE\_INVALID.

---

**servo→get(userData)****YServo****servo→userData()servo.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**servo→isOnline()servo.isOnline()****YServo**

Checks if the servo is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the servo in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the servo.

**Returns :**

`true` if the servo can be reached, and `false` otherwise

**servo→load()servo.load()****YServo**

Preloads the servo cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**servo→move()servo.move()****YServo**

Performs a smooth move at constant speed toward a given position.

```
def move( target, ms_duration)
```

**Parameters :**

**target** new position at the end of the move

**ms\_duration** total duration of the move, in milliseconds

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**servo→nextServo()servo.nextServo()****YServo**

Continues the enumeration of servos started using `yFirstServo()`.

```
def nextServo( )
```

**Returns :**

a pointer to a `YServo` object, corresponding to a servo currently online, or a `null` pointer if there are no more servos to enumerate.

**servo→registerValueCallback()**  
**servo.registerValueCallback()****YServo**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

---

<b>servo→set_enabled()</b>	<b>YServo</b>
<b>servo→setEnabled()servo.set_enabled()</b>	

---

Stops or starts the servo.

```
def set_enabled( newval)
```

**Parameters :**

**newval** either Y\_ENABLED\_FALSE or Y\_ENABLED\_TRUE

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**servo→set\_enabledAtPowerOn()**  
**servo→setEnabledAtPowerOn()**  
**servo.set\_enabledAtPowerOn()**

**YServo**

Configure the servo signal generator state at power up.

```
def set_enabledAtPowerOn( newval)
```

Remember to call the matching module saveToFlash( ) method, otherwise this call will have no effect.

**Parameters :**

**newval** either Y\_ENABLEDATPOWERON\_FALSE or Y\_ENABLEDATPOWERON\_TRUE

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**servo→set\_logicalName()****YServo****servo→setLogicalName()servo.set\_logicalName()**

Changes the logical name of the servo.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the servo.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**servo→set\_neutral()  
servo→setNeutral()servo.set\_neutral()****YServo**

Changes the duration of the pulse corresponding to the neutral position of the servo.

```
def set_neutral( newval )
```

The duration is specified in microseconds, and the standard value is 1500 [us]. This setting makes it possible to shift the range of use of the servo. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

**Parameters :**

**newval** an integer corresponding to the duration of the pulse corresponding to the neutral position of the servo

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

<b>servo→set_position()</b>	<b>YServo</b>
<b>servo→setPosition()servo.set_position()</b>	

---

Changes immediately the servo driving position.

```
def set_position( newval)
```

**Parameters :**

**newval** an integer corresponding to immediately the servo driving position

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**servo→set\_positionAtPowerOn()**  
**servo→setPositionAtPowerOn()**  
**servo.set\_positionAtPowerOn()**

**YServo**

Configure the servo position at device power up.

```
def set_positionAtPowerOn( newval)
```

Remember to call the matching module saveToFlash( ) method, otherwise this call will have no effect.

**Parameters :**

**newval** an integer

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**servo→set\_range()****YServo****servo→setRange()servo.set\_range()**

Changes the range of use of the servo, specified in per cents.

```
def set_range( newval)
```

A range of 100% corresponds to a standard control signal, that varies from 1 [ms] to 2 [ms]. When using a servo that supports a double range, from 0.5 [ms] to 2.5 [ms], you can select a range of 200%. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

**Parameters :**

**newval** an integer corresponding to the range of use of the servo, specified in per cents

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**servo→set(userData)**

**YServo**

**servo→setUserData()servo.set.userData()**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.38. Temperature function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_temperature.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YTemperature = yoctolib.YTemperature;
php	require_once('yocto_temperature.php');
cpp	#include "yocto_temperature.h"
m	#import "yocto_temperature.h"
pas	uses yocto_temperature;
vb	yocto_temperature.vb
cs	yocto_temperature.cs
java	import com.yoctopuce.YoctoAPI.YTemperature;
py	from yocto_temperature import *

### Global functions

#### yFindTemperature(func)

Retrieves a temperature sensor for a given identifier.

#### yFirstTemperature()

Starts the enumeration of temperature sensors currently accessible.

### YTemperature methods

#### temperature→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### temperature→describe()

Returns a short text that describes unambiguously the instance of the temperature sensor in the form TYPE (NAME) = SERIAL . FUNCTIONID.

#### temperature→get\_advertisedValue()

Returns the current value of the temperature sensor (no more than 6 characters).

#### temperature→get\_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### temperature→get\_currentValue()

Returns the current value of the temperature.

#### temperature→get\_errorMessage()

Returns the error message of the latest error with the temperature sensor.

#### temperature→get\_errorType()

Returns the numerical error code of the latest error with the temperature sensor.

#### temperature→get\_friendlyName()

Returns a global identifier of the temperature sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### temperature→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### temperature→get\_functionId()

Returns the hardware identifier of the temperature sensor, without reference to the module.

#### temperature→get\_hardwareId()

Returns the unique hardware identifier of the temperature sensor in the form SERIAL . FUNCTIONID.

<b>temperature→get_highestValue()</b>	Returns the maximal value observed for the temperature since the device was started.
<b>temperature→get_logFrequency()</b>	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>temperature→get_logicalName()</b>	Returns the logical name of the temperature sensor.
<b>temperature→get_lowestValue()</b>	Returns the minimal value observed for the temperature since the device was started.
<b>temperature→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>temperature→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>temperature→get_recordedData(startTime, endTime)</b>	Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>temperature→get_reportFrequency()</b>	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>temperature→get_resolution()</b>	Returns the resolution of the measured values.
<b>temperature→get_sensorType()</b>	Returns the temperature sensor type.
<b>temperature→get_unit()</b>	Returns the measuring unit for the temperature.
<b>temperature→get(userData)</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>temperature→isOnline()</b>	Checks if the temperature sensor is currently reachable, without raising any error.
<b>temperature→isOnline_async(callback, context)</b>	Checks if the temperature sensor is currently reachable, without raising any error (asynchronous version).
<b>temperature→load(msValidity)</b>	Preloads the temperature sensor cache with a specified validity duration.
<b>temperature→loadCalibrationPoints(rawValues, refValues)</b>	Retrieves error correction data points previously entered using the method calibrateFromPoints.
<b>temperature→load_async(msValidity, callback, context)</b>	Preloads the temperature sensor cache with a specified validity duration (asynchronous version).
<b>temperature→nextTemperature()</b>	Continues the enumeration of temperature sensors started using yFirstTemperature( ).
<b>temperature→registerTimedReportCallback(callback)</b>	Registers the callback function that is invoked on every periodic timed notification.
<b>temperature→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>temperature→set_highestValue(newval)</b>	Changes the recorded maximal value observed.
<b>temperature→set_logFrequency(newval)</b>	Changes the datalogger recording frequency for this function.

**temperature→set\_logicalName(newval)**

Changes the logical name of the temperature sensor.

**temperature→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**temperature→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**temperature→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**temperature→set\_sensorType(newval)**

Modify the temperature sensor type.

**temperature→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**temperature→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YTemperature.FindTemperature() yFindTemperature()YTemperature.FindTemperature()

Retrieves a temperature sensor for a given identifier.

```
def FindTemperature( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the temperature sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTemperature.isOnline()` to test if the temperature sensor is indeed online at a given time. In case of ambiguity when looking for a temperature sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the temperature sensor

### Returns :

a `YTemperature` object allowing you to drive the temperature sensor.

**YTemperature.FirstTemperature()****YTemperature****yFirstTemperature()YTemperature.FirstTemperature()**

Starts the enumeration of temperature sensors currently accessible.

```
def FirstTemperature( )
```

Use the method `YTemperature.nextTemperature( )` to iterate on next temperature sensors.

**Returns :**

a pointer to a `YTemperature` object, corresponding to the first temperature sensor currently online, or a null pointer if there are none.

**temperature→calibrateFromPoints()**  
**temperature.calibrateFromPoints()****YTemperature**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**temperature→describe()temperature.describe()****YTemperature**

Returns a short text that describes unambiguously the instance of the temperature sensor in the form TYPE ( NAME )=SERIAL . FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

```
a string that describes the temperature sensor (ex:  
Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)
```

**temperature→get\_advertisedValue()**  
**temperature→advertisedValue()**  
**temperature.get\_advertisedValue()**

**YTemperature**

---

Returns the current value of the temperature sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the temperature sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**temperature→get\_currentRawValue()**  
**temperature→currentRawValue()**  
**temperature.get\_currentRawValue()**

**YTemperature**

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**temperature→get\_currentValue()**  
**temperature→currentValue()**  
**temperature.get\_currentValue()**

---

**YTemperature**

Returns the current value of the temperature.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the temperature

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

**temperature→getErrorMessage()**  
**temperature→errorMessage()**  
**temperature.getErrorMessage()**

**YTemperature**

Returns the error message of the latest error with the temperature sensor.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the temperature sensor object

**temperature→get\_errorType()**  
**temperature→errorType()**  
**temperature.get\_errorType()**

---

**YTemperature**

Returns the numerical error code of the latest error with the temperature sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the temperature sensor object

**temperature→get\_friendlyName()**  
**temperature→friendlyName()**  
**temperature.get\_friendlyName()**

**YTemperature**

---

Returns a global identifier of the temperature sensor in the format MODULE\_NAME.FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the temperature sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the temperature sensor (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the temperature sensor using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**temperature→get\_functionDescriptor()**  
**temperature→functionDescriptor()**  
**temperature.get\_functionDescriptor()**

**YTemperature**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**temperature→get\_functionId()**  
**temperature→functionId()**  
**temperature.get\_functionId()**

**YTemperature**

Returns the hardware identifier of the temperature sensor, without reference to the module.

```
def get_functionId( )
```

For example relay1

**Returns :**

a string that identifies the temperature sensor (ex: relay1) On failure, throws an exception or returns Y\_FUNCTIONID\_INVALID.

**temperature→get\_hardwareId()**  
**temperature→hardwareId()**  
**temperature.get\_hardwareId()**

**YTemperature**

Returns the unique hardware identifier of the temperature sensor in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the temperature sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the temperature sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**temperature→get\_highestValue()**  
**temperature→highestValue()**  
**temperature.get\_highestValue()**

**YTemperature**

Returns the maximal value observed for the temperature since the device was started.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the temperature since the device was started

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

---

<b>temperature→get_logFrequency()</b>	<b>YTemperature</b>
<b>temperature→logFrequency()</b>	
<b>temperature.get_logFrequency()</b>	

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**temperature→get\_logicalName()**  
**temperature→logicalName()**  
**temperature.get\_logicalName()**

**YTemperature**

Returns the logical name of the temperature sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the temperature sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**temperature→get\_lowestValue()**  
**temperature→lowestValue()**  
**temperature.get\_lowestValue()**

**YTemperature**

Returns the minimal value observed for the temperature since the device was started.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the temperature since the device was started

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**temperature→get\_module()****YTemperature****temperature→module()temperature.get\_module()**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**temperature→get\_recordedData()**  
**temperature→recordedData()**  
**temperature.get\_recordedData()**

**YTemperature**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime )
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**temperature→get\_reportFrequency()**  
**temperature→reportFrequency()**  
**temperature.get\_reportFrequency()**

**YTemperature**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**temperature→get\_resolution()**  
**temperature→resolution()**  
**temperature.get\_resolution()**

---

**YTemperature**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**temperature→get\_sensorType()**  
**temperature→sensorType()**  
**temperature.get\_sensorType()**

**YTemperature**

Returns the temperature sensor type.

```
def get_sensorType( )
```

**Returns :**

a value among Y\_SENSORTYPE\_DIGITAL, Y\_SENSORTYPE\_TYPE\_K,  
Y\_SENSORTYPE\_TYPE\_E, Y\_SENSORTYPE\_TYPE\_J, Y\_SENSORTYPE\_TYPE\_N,  
Y\_SENSORTYPE\_TYPE\_R, Y\_SENSORTYPE\_TYPE\_S, Y\_SENSORTYPE\_TYPE\_T,  
Y\_SENSORTYPE\_PT100\_4WIRES, Y\_SENSORTYPE\_PT100\_3WIRES and  
Y\_SENSORTYPE\_PT100\_2WIRES corresponding to the temperature sensor type

On failure, throws an exception or returns Y\_SENSORTYPE\_INVALID.

**temperature→get\_unit()**

**YTemperature**

**temperature→unit()temperature.get\_unit()**

---

Returns the measuring unit for the temperature.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the temperature

On failure, throws an exception or returns Y\_UNIT\_INVALID.

---

**temperature→get(userData)****YTemperature****temperature→userData()temperature.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**temperature→isOnline()****YTemperature**

Checks if the temperature sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the temperature sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the temperature sensor.

**Returns :**

`true` if the temperature sensor can be reached, and `false` otherwise

**temperature→load()temperature.load()****YTemperature**

Preloads the temperature sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**temperature→loadCalibrationPoints()**  
**temperature.loadCalibrationPoints()****YTemperature**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**temperature→nextTemperature()**  
**temperature.nextTemperature()**

**YTemperature**

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

```
def nextTemperature( )
```

**Returns :**

a pointer to a `YTemperature` object, corresponding to a temperature sensor currently online, or a `null` pointer if there are no more temperature sensors to enumerate.

**temperature→registerTimedReportCallback()  
temperature.registerTimedReportCallback()****YTemperature**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**temperature→registerValueCallback()  
temperature.registerValueCallback()****YTemperature**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**temperature→set\_highestValue()**  
**temperature→setHighestValue()**  
**temperature.set\_highestValue()**

**YTemperature**

Changes the recorded maximal value observed.

```
def set_highestValue( newval )
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**temperature→set\_logFrequency()**  
**temperature→setLogFrequency()**  
**temperature.set\_logFrequency()**

**YTemperature**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**temperature→set\_logicalName()**  
**temperature→setLogicalName()**  
**temperature.set\_logicalName()**

**YTemperature**

Changes the logical name of the temperature sensor.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the temperature sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**temperature→set\_lowestValue()**  
**temperature→setLowestValue()**  
**temperature.set\_lowestValue()**

YTemperature

Changes the recorded minimal value observed.

```
def set_lowestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**temperature→set\_reportFrequency()**  
**temperature→setReportFrequency()**  
**temperature.set\_reportFrequency()**

**YTemperature**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**temperature→set\_resolution()**  
**temperature→setResolution()**  
**temperature.set\_resolution()**

**YTemperature**

Changes the resolution of the measured physical values.

```
def set_resolution( newval )
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**temperature→set\_sensorType()**  
**temperature→setSensorType()**  
**temperature.set\_sensorType()**

**YTemperature**

Modify the temperature sensor type.

```
def set_sensorType( newval )
```

This function is used to define the type of thermocouple (K,E...) used with the device. This will have no effect if module is using a digital sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a value among `Y_SENSORTYPE_DIGITAL`, `Y_SENSORTYPE_TYPE_K`,  
`Y_SENSORTYPE_TYPE_E`, `Y_SENSORTYPE_TYPE_J`, `Y_SENSORTYPE_TYPE_N`,  
`Y_SENSORTYPE_TYPE_R`, `Y_SENSORTYPE_TYPE_S`, `Y_SENSORTYPE_TYPE_T`,  
`Y_SENSORTYPE_PT100_4WIRES`, `Y_SENSORTYPE_PT100_3WIRES` and  
`Y_SENSORTYPE_PT100_2WIRES`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**temperature→set(userData)**  
**temperature→setUserData()**  
**temperature.set(userData)**

**YTemperature**

Stores a user context provided as argument in the userData attribute of the function.

```
def setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.39. Tilt function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_tilt.js'></script>
nodejs var yoctolib = require('yoctolib');
var YTilt = yoctolib.YTilt;
php require_once('yocto_tilt.php');
cpp #include "yocto_tilt.h"
m #import "yocto_tilt.h"
pas uses yocto_tilt;
vb yocto_tilt.vb
cs yocto_tilt.cs
java import com.yoctopuce.YoctoAPI.YTilt;
py from yocto_tilt import *

```

### Global functions

#### yFindTilt(func)

Retrieves a tilt sensor for a given identifier.

#### yFirstTilt()

Starts the enumeration of tilt sensors currently accessible.

### YTilt methods

#### tilt→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### tilt→describe()

Returns a short text that describes unambiguously the instance of the tilt sensor in the form TYPE (NAME) = SERIAL . FUNCTIONID.

#### tilt→get\_advertisedValue()

Returns the current value of the tilt sensor (no more than 6 characters).

#### tilt→get\_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### tilt→get\_currentValue()

Returns the current value of the inclination.

#### tilt→get\_errorMessage()

Returns the error message of the latest error with the tilt sensor.

#### tilt→get\_errorType()

Returns the numerical error code of the latest error with the tilt sensor.

#### tilt→get\_friendlyName()

Returns a global identifier of the tilt sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### tilt→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### tilt→get\_functionId()

Returns the hardware identifier of the tilt sensor, without reference to the module.

#### tilt→get\_hardwareId()

Returns the unique hardware identifier of the tilt sensor in the form SERIAL . FUNCTIONID.

<b>tilt→get_highestValue()</b>	Returns the maximal value observed for the inclination since the device was started.
<b>tilt→get_logFrequency()</b>	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
<b>tilt→get_logicalName()</b>	Returns the logical name of the tilt sensor.
<b>tilt→get_lowestValue()</b>	Returns the minimal value observed for the inclination since the device was started.
<b>tilt→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>tilt→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>tilt→get_recordedData(startTime, endTime)</b>	Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
<b>tilt→get_reportFrequency()</b>	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
<b>tilt→get_resolution()</b>	Returns the resolution of the measured values.
<b>tilt→get_unit()</b>	Returns the measuring unit for the inclination.
<b>tilt→get(userData)</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>tilt→isOnline()</b>	Checks if the tilt sensor is currently reachable, without raising any error.
<b>tilt→isOnline_async(callback, context)</b>	Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).
<b>tilt→load(msValidity)</b>	Preloads the tilt sensor cache with a specified validity duration.
<b>tilt→loadCalibrationPoints(rawValues, refValues)</b>	Retrieves error correction data points previously entered using the method calibrateFromPoints.
<b>tilt→load_async(msValidity, callback, context)</b>	Preloads the tilt sensor cache with a specified validity duration (asynchronous version).
<b>tilt→nextTilt()</b>	Continues the enumeration of tilt sensors started using yFirstTilt( ).
<b>tilt→registerTimedReportCallback(callback)</b>	Registers the callback function that is invoked on every periodic timed notification.
<b>tilt→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>tilt→set_highestValue(newval)</b>	Changes the recorded maximal value observed.
<b>tilt→set_logFrequency(newval)</b>	Changes the datalogger recording frequency for this function.
<b>tilt→set_logicalName(newval)</b>	Changes the logical name of the tilt sensor.

### 3. Reference

---

**`tilt→set_lowestValue(newval)`**

Changes the recorded minimal value observed.

**`tilt→set_reportFrequency(newval)`**

Changes the timed value notification frequency for this function.

**`tilt→set_resolution(newval)`**

Changes the resolution of the measured physical values.

**`tilt→set_userData(data)`**

Stores a user context provided as argument in the userData attribute of the function.

**`tilt→wait_async(callback, context)`**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**YTilt.FindTilt()****YTilt****yFindTilt()YTilt.FindTilt()**

Retrieves a tilt sensor for a given identifier.

```
def FindTilt( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the tilt sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YTilt.isOnline()` to test if the tilt sensor is indeed online at a given time. In case of ambiguity when looking for a tilt sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the tilt sensor

**Returns :**

a `YTilt` object allowing you to drive the tilt sensor.

## **YTilt.FirstTilt()**

**YTilt**

### **yFirstTilt()YTilt.FirstTilt()**

Starts the enumeration of tilt sensors currently accessible.

```
def FirstTilt( )
```

Use the method `YTilt.nextTilt()` to iterate on next tilt sensors.

**Returns :**

a pointer to a `YTilt` object, corresponding to the first tilt sensor currently online, or a `null` pointer if there are none.

**tilt→calibrateFromPoints()tilt.calibrateFromPoints()****YTilt**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**tilt→describe()tilt.describe()****YTilt**

Returns a short text that describes unambiguously the instance of the tilt sensor in the form  
TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the tilt sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

---

**tilt→get\_advertisedValue()****YTilt****tilt→advertisedValue()tilt.get\_advertisedValue()**

Returns the current value of the tilt sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the tilt sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**tilt→get\_currentRawValue()** YTilt  
**tilt→currentRawValue()tilt.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**tilt→get\_currentValue()**

YTilt

**tilt→currentValue()tilt.get\_currentValue()**

Returns the current value of the inclination.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the inclination

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

**tilt→get\_errorMessage()** YTilt  
**tilt→errorMessage()tilt.getErrorMessage()**

---

Returns the error message of the latest error with the tilt sensor.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the tilt sensor object

---

**tilt→get\_errorType()****YTilt****tilt→errorType()tilt.get\_errorType()**

Returns the numerical error code of the latest error with the tilt sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the tilt sensor object

---

<b>tilt→get_friendlyName()</b>	<b>YTilt</b>
<b>tilt→friendlyName()tilt.get_friendlyName()</b>	

---

Returns a global identifier of the tilt sensor in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the tilt sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the tilt sensor (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the tilt sensor using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**tilt→get\_functionDescriptor()****YTilt****tilt→functionDescriptor()tilt.get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**tilt→get\_functionId()**

YTilt

**tilt→functionId()tilt.get\_functionId()**

---

Returns the hardware identifier of the tilt sensor, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the tilt sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

---

**tilt→get\_hardwareId()****YTilt****tilt→hardwareId()tilt.get\_hardwareId()**

Returns the unique hardware identifier of the tilt sensor in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the tilt sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the tilt sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**tilt→get\_highestValue()**

**YTilt**

**tilt→highestValue()tilt.get\_highestValue()**

---

Returns the maximal value observed for the inclination since the device was started.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the inclination since the device was started

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**tilt→get\_logFrequency()****YTilt****tilt→logFrequency()tilt.get\_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

<b>tilt→get_logicalName()</b>	<b>YTilt</b>
<b>tilt→logicalName()tilt.get_logicalName()</b>	

---

Returns the logical name of the tilt sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the tilt sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**tilt→get\_lowestValue()****YTilt****tilt→lowestValue()tilt.get\_lowestValue()**

Returns the minimal value observed for the inclination since the device was started.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the inclination since the device was started

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**tilt→get\_module()  
tilt→module()tilt.get\_module()**

---

YTilt

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**tilt→get\_recordedData()**

YTilt

**tilt→recordedData(tilt.get\_recordedData())**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

**Parameters :**

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

**Returns :**

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

---

<b>tilt→get_reportFrequency()</b>	<b>YTilt</b>
<b>tilt→reportFrequency()tilt.get_reportFrequency()</b>	

---

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**tilt→get\_resolution()**

YTilt

**tilt→resolution()tilt.get\_resolution()**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**tilt→get\_unit()**  
**tilt→unit()tilt.get\_unit()**

---

YTilt

Returns the measuring unit for the inclination.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the inclination

On failure, throws an exception or returns Y\_UNIT\_INVALID.

---

**tilt→get(userData)****YTilt****tilt→userData()tilt.get(userData)**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**tilt→isOnline()  
tilt.isOnline()****YTilt**

Checks if the tilt sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the tilt sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the tilt sensor.

**Returns :**

`true` if the tilt sensor can be reached, and `false` otherwise

**tilt→load()tilt.load()****YTilt**

Preloads the tilt sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**tilt→loadCalibrationPoints()**  
**tilt.loadCalibrationPoints()****YTilt**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**tilt→nextTilt()tilt.nextTilt()****YTilt**

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

```
def nextTilt( )
```

**Returns :**

a pointer to a `YTilt` object, corresponding to a tilt sensor currently online, or a null pointer if there are no more tilt sensors to enumerate.

**tilt→registerTimedReportCallback()  
tilt.registerTimedReportCallback()****YTilt**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**tilt→registerValueCallback()**  
**tilt.registerValueCallback()**

YTilt

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

---

<b>tilt→set_highestValue()</b>	<b>YTilt</b>
<b>tilt→setHighestValue()tilt.set_highestValue()</b>	

---

Changes the recorded maximal value observed.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**tilt→set\_logFrequency()**

YTilt

**tilt→setLogFrequency()tilt.set\_logFrequency()**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

<b>tilt→set_logicalName()</b>	<b>YTilt</b>
<b>tilt→setLogicalName()tilt.set_logicalName()</b>	

---

Changes the logical name of the tilt sensor.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the tilt sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**tilt→set\_lowestValue()**

YTilt

**tilt→setLowestValue()tilt.set\_lowestValue()**

Changes the recorded minimal value observed.

```
def set_lowestValue( newval )
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**tilt→set\_reportFrequency()** YTilt  
**tilt→setReportFrequency()tilt.set\_reportFrequency()**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**tilt→set\_resolution()****YTilt****tilt→setResolution()tilt.set\_resolution()**

Changes the resolution of the measured physical values.

```
def set_resolution( newval )
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**tilt→set(userData)**

YTilt

**tilt→setUserData()tilt.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.40. Voc function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_voc.js'></script>
node.js	var yoctolib = require('yoctolib');
	var YVoc = yoctolib.YVoc;
php	require_once('yocto_voc.php');
cpp	#include "yocto_voc.h"
m	#import "yocto_voc.h"
pas	uses yocto_voc;
vb	yocto_voc.vb
cs	yocto_voc.cs
java	import com.yoctopuce.YoctoAPI.YVoc;
py	from yocto_voc import *

### Global functions

#### yFindVoc(func)

Retrieves a Volatile Organic Compound sensor for a given identifier.

#### yFirstVoc()

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

### YVoc methods

#### voc→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### voc→describe()

Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form TYPE ( NAME ) = SERIAL . FUNCTIONID.

#### voc→get\_advertisedValue()

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

#### voc→get\_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

#### voc→get\_currentValue()

Returns the current measure for the estimated VOC concentration.

#### voc→get\_errorMessage()

Returns the error message of the latest error with the Volatile Organic Compound sensor.

#### voc→get\_errorType()

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

#### voc→get\_friendlyName()

Returns a global identifier of the Volatile Organic Compound sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### voc→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### voc→get\_functionId()

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

#### voc→get\_hardwareId()

### 3. Reference

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form SERIAL.FUNCTIONID.

#### voc->get\_highestValue()

Returns the maximal value observed for the estimated VOC concentration.

#### voc->get\_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

#### voc->get\_logicalName()

Returns the logical name of the Volatile Organic Compound sensor.

#### voc->get\_lowestValue()

Returns the minimal value observed for the estimated VOC concentration.

#### voc->get\_module()

Gets the YModule object for the device on which the function is located.

#### voc->get\_module\_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

#### voc->get\_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

#### voc->get\_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

#### voc->get\_resolution()

Returns the resolution of the measured values.

#### voc->get\_unit()

Returns the measuring unit for the estimated VOC concentration.

#### voc->get\_userData()

Returns the value of the userData attribute, as previously stored using method set(userData).

#### voc->isOnline()

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

#### voc->isOnline\_async(callback, context)

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error (asynchronous version).

#### voc->load(msValidity)

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

#### voc->loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

#### voc->load\_async(msValidity, callback, context)

Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).

#### voc->nextVoc()

Continues the enumeration of Volatile Organic Compound sensors started using yFirstVoc( ).

#### voc->registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

#### voc->registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

#### voc->set\_highestValue(newval)

Changes the recorded maximal value observed for the estimated VOC concentration.

**voc→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**voc→set\_logicalName(newval)**

Changes the logical name of the Volatile Organic Compound sensor.

**voc→set\_lowestValue(newval)**

Changes the recorded minimal value observed for the estimated VOC concentration.

**voc→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**voc→set\_resolution(newval)**

Changes the resolution of the measured physical values.

**voc→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**voc→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YVoc.FindVoc() yFindVoc()YVoc.FindVoc()

YVoc

Retrieves a Volatile Organic Compound sensor for a given identifier.

```
def FindVoc( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the Volatile Organic Compound sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoc.isOnline()` to test if the Volatile Organic Compound sensor is indeed online at a given time. In case of ambiguity when looking for a Volatile Organic Compound sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the Volatile Organic Compound sensor

### Returns :

a `YVoc` object allowing you to drive the Volatile Organic Compound sensor.

**YVoc.FirstVoc()****YVoc****yFirstVoc()YVoc.FirstVoc()**

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

```
def FirstVoc( )
```

Use the method `YVoc.nextVoc()` to iterate on next Volatile Organic Compound sensors.

**Returns :**

a pointer to a `YVoc` object, corresponding to the first Volatile Organic Compound sensor currently online, or a null pointer if there are none.

**voc→calibrateFromPoints()|voc.calibrateFromPoints()****YVoc**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voc→describe()voc.describe()****YVoc**

Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form TYPE (NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

```
a string that describes the Volatile Organic Compound sensor (ex:  
Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)
```

**voc→get\_advertisedValue()****YVoc****voc→advertisedValue()voc.get\_advertisedValue()**

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the Volatile Organic Compound sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

---

**voc→get\_currentRawValue()****YVoc****voc→currentRawValue()voc.get\_currentRawValue()**

---

Returns the unrounded and uncalibrated raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

**voc→get\_currentValue()** YVoc  
**voc→currentValue()voc.get\_currentValue()**

---

Returns the current measure for the estimated VOC concentration.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current measure for the estimated VOC concentration

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

---

**voc→get\_errorMessage()****YVoc****voc→errorMessage()voc.get\_errorMessage()**

Returns the error message of the latest error with the Volatile Organic Compound sensor.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the Volatile Organic Compound sensor object

**voc→get\_errorType()**  
**voc→errorType()voc.get\_errorType()**

---

**YVoc**

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the Volatile Organic Compound sensor object

**voc→get\_friendlyName()****YVoc****voc→friendlyName()voc.get\_friendlyName()**

Returns a global identifier of the Volatile Organic Compound sensor in the format MODULE\_NAME.FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the Volatile Organic Compound sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the Volatile Organic Compound sensor (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the Volatile Organic Compound sensor using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**voc->get\_functionDescriptor()**  
**voc->functionDescriptor()**  
**voc.get\_functionDescriptor()**

**YVoc**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**voc→get\_functionId()****YVoc****voc→functionId()voc.get\_functionId()**

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the Volatile Organic Compound sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

---

<b>voc→get_hardwareId()</b>	<b>YVoc</b>
<b>voc→hardwareId()voc.get_hardwareId()</b>	

---

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the Volatile Organic Compound sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the Volatile Organic Compound sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

---

**voc→get\_highestValue()****YVoc****voc→highestValue()voc.get\_highestValue()**

Returns the maximal value observed for the estimated VOC concentration.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the estimated VOC concentration

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**voc→get\_logFrequency()** YVoc  
**voc→logFrequency()voc.get\_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**voc→get\_logicalName()****YVoc****voc→logicalName()voc.get\_logicalName()**

Returns the logical name of the Volatile Organic Compound sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the Volatile Organic Compound sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**voc→get\_lowestValue()**

**YVoc**

**voc→lowestValue()voc.get\_lowestValue()**

---

Returns the minimal value observed for the estimated VOC concentration.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the estimated VOC concentration

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**voc→get\_module()****YVoc****voc→module()voc.get\_module()**

Gets the YModule object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

**Returns :**

an instance of YModule

---

<b>voc→get_recordedData()</b>	<b>YVoc</b>
<b>voc→recordedData()voc.get_recordedData()</b>	

---

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime )
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

#### Parameters :

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

#### Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**voc→get\_reportFrequency()****YVoc****voc→reportFrequency()voc.get\_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**voc→get\_resolution()**  
**voc→resolution()voc.get\_resolution()**

---

**YVoc**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**voc→get\_unit()****YVoc****voc→unit()voc.get\_unit()**

Returns the measuring unit for the estimated VOC concentration.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the estimated VOC concentration

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**voc→get(userData)**

**YVoc**

**voc→userData()voc.get(userData())**

---

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**voc→isOnline()voc.isOnline()****YVoc**

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the Volatile Organic Compound sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the Volatile Organic Compound sensor.

**Returns :**

`true` if the Volatile Organic Compound sensor can be reached, and `false` otherwise

**voc→load()voc.load()****YVoc**

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**voc→loadCalibrationPoints()  
voc.loadCalibrationPoints()****YVoc**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## voc→nextVoc()voc.nextVoc()

YVoc

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

```
def nextVoc( )
```

**Returns :**

a pointer to a `YVoc` object, corresponding to a Volatile Organic Compound sensor currently online, or a null pointer if there are no more Volatile Organic Compound sensors to enumerate.

**voc→registerTimedReportCallback()**  
**voc.registerTimedReportCallback()****YVoc**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**voc→registerValueCallback()**  
**voc.registerValueCallback()****YVoc**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**voc→set\_highestValue()****YVoc****voc→setHighestValue()voc.set\_highestValue()**

Changes the recorded maximal value observed for the estimated VOC concentration.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed for the estimated VOC concentration

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voc→set\_logFrequency()** YVoc  
**voc→setLogFrequency()voc.set\_logFrequency()**

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voc→set\_logicalName()****YVoc****voc→setLogicalName()voc.set\_logicalName()**

Changes the logical name of the Volatile Organic Compound sensor.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the Volatile Organic Compound sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**voc→set\_lowestValue()** YVoc  
**voc→setLowestValue()voc.set\_lowestValue()**

Changes the recorded minimal value observed for the estimated VOC concentration.

```
def set_lowestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed for the estimated VOC concentration

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voc→set\_reportFrequency()**  
**voc→setReportFrequency()**  
**voc.set\_reportFrequency()**

YVoc

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voc→set\_resolution()  
voc→setResolution()voc.set\_resolution()****YVoc**

Changes the resolution of the measured physical values.

```
def set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured physical values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

---

**voc→set(userData)****YVoc****voc→setUserData()voc.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.41. Voltage function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_voltage.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YVoltage = yoctolib.YVoltage;
php	require_once('yocto_voltage.php');
cpp	#include "yocto_voltage.h"
m	#import "yocto_voltage.h"
pas	uses yocto_voltage;
vb	yocto_voltage.vb
cs	yocto_voltage.cs
java	import com.yoctopuce.YoctoAPI.YVoltage;
py	from yocto_voltage import *

### Global functions

#### yFindVoltage(func)

Retrieves a voltage sensor for a given identifier.

#### yFirstVoltage()

Starts the enumeration of voltage sensors currently accessible.

### YVoltage methods

#### voltage→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

#### voltage→describe()

Returns a short text that describes unambiguously the instance of the voltage sensor in the form TYPE (NAME) = SERIAL . FUNCTIONID.

#### voltage→get\_advertisedValue()

Returns the current value of the voltage sensor (no more than 6 characters).

#### voltage→get\_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

#### voltage→get\_currentValue()

Returns the current measure for the voltage.

#### voltage→get\_errorMessage()

Returns the error message of the latest error with the voltage sensor.

#### voltage→get\_errorType()

Returns the numerical error code of the latest error with the voltage sensor.

#### voltage→get\_friendlyName()

Returns a global identifier of the voltage sensor in the format MODULE\_NAME . FUNCTION\_NAME.

#### voltage→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### voltage→get\_functionId()

Returns the hardware identifier of the voltage sensor, without reference to the module.

#### voltage→get\_hardwareId()

Returns the unique hardware identifier of the voltage sensor in the form SERIAL . FUNCTIONID.

**voltage→get\_highestValue()**

Returns the maximal value observed for the voltage.

**voltage→get\_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

**voltage→get\_logicalName()**

Returns the logical name of the voltage sensor.

**voltage→get\_lowestValue()**

Returns the minimal value observed for the voltage.

**voltage→get\_module()**

Gets the YModule object for the device on which the function is located.

**voltage→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**voltage→get\_recordedData(startTime, endTime)**

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

**voltage→get\_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**voltage→get\_resolution()**

Returns the resolution of the measured values.

**voltage→get\_unit()**

Returns the measuring unit for the voltage.

**voltage→get(userData)**

Returns the value of the userData attribute, as previously stored using method set(userData).

**voltage→isOnline()**

Checks if the voltage sensor is currently reachable, without raising any error.

**voltage→isOnline\_async(callback, context)**

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

**voltage→load(msValidity)**

Preloads the voltage sensor cache with a specified validity duration.

**voltage→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

**voltage→load\_async(msValidity, callback, context)**

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

**voltage→nextVoltage()**

Continues the enumeration of voltage sensors started using yFirstVoltage( ).

**voltage→registerTimedReportCallback(callback)**

Registers the callback function that is invoked on every periodic timed notification.

**voltage→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**voltage→set\_highestValue(newval)**

Changes the recorded maximal value observed pour the voltage.

**voltage→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**voltage→set\_logicalName(newval)**

Changes the logical name of the voltage sensor.

### 3. Reference

---

**voltage→set\_lowestValue(newval)**

Changes the recorded minimal value observed pour the voltage.

**voltage→set\_reportFrequency(newval)**

Changes the timed value notification frequency for this function.

**voltage→set\_resolution(newval)**

Changes the resolution of the measured values.

**voltage→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**voltage→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**YVoltage.FindVoltage()****YVoltage****yFindVoltage()YVoltage.FindVoltage()**

Retrieves a voltage sensor for a given identifier.

```
def FindVoltage( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltage.isOnline()` to test if the voltage sensor is indeed online at a given time. In case of ambiguity when looking for a voltage sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the voltage sensor

**Returns :**

a `YVoltage` object allowing you to drive the voltage sensor.

## **YVoltage.FirstVoltage() yFirstVoltage()YVoltage.FirstVoltage()**

---

**YVoltage**

Starts the enumeration of voltage sensors currently accessible.

```
def FirstVoltage( )
```

Use the method `YVoltage.nextVoltage( )` to iterate on next voltage sensors.

**Returns :**

a pointer to a `YVoltage` object, corresponding to the first voltage sensor currently online, or a null pointer if there are none.

**voltage→calibrateFromPoints()**  
**voltage.calibrateFromPoints()****YVoltage**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
def calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact [support@yoctopuce.com](mailto:support@yoctopuce.com).

**Parameters :**

**rawValues** array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

**refValues** array of floating point numbers, corresponding to the corrected values for the correction points.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voltage→describe()voltage.describe()****YVoltage**

Returns a short text that describes unambiguously the instance of the voltage sensor in the form  
TYPE ( NAME )=SERIAL . FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the voltage sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

**voltage→get\_advertisedValue()**  
**voltage→advertisedValue()**  
**voltage.get\_advertisedValue()**

**YVoltage**

Returns the current value of the voltage sensor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the voltage sensor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**voltage→get\_currentRawValue()**  
**voltage→currentRawValue()**  
**voltage.get\_currentRawValue()**

**YVoltage**

---

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns Y\_CURRENTRAWVALUE\_INVALID.

---

<b>voltage→get_currentValue()</b>	<b>YVoltage</b>
<b>voltage→currentValue()voltage.get_currentValue()</b>	

---

Returns the current measure for the voltage.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current measure for the voltage

On failure, throws an exception or returns Y\_CURRENTVALUE\_INVALID.

**voltage→get\_errorMessage()** **YVoltage**  
**voltage→errorMessage()voltage.get\_errorMessage()**

---

Returns the error message of the latest error with the voltage sensor.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the voltage sensor object

---

**voltage→get\_errorType()****YVoltage****voltage→errorType()voltage.get\_errorType()**

Returns the numerical error code of the latest error with the voltage sensor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the voltage sensor object

---

<b>voltage→get_friendlyName()</b>	<b>YVoltage</b>
<b>voltage→friendlyName()voltage.get_friendlyName()</b>	

---

Returns a global identifier of the voltage sensor in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the voltage sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage sensor (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the voltage sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**voltage→get\_functionDescriptor()**  
**voltage→functionDescriptor()**  
**voltage.get\_functionDescriptor()**

**YVoltage**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**voltage→get\_functionId()**

**YVoltage**

**voltage→functionId()voltage.get\_functionId()**

---

Returns the hardware identifier of the voltage sensor, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the voltage sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**voltage→get\_hardwareId()****YVoltage****voltage→hardwareId()voltage.get\_hardwareId()**

Returns the unique hardware identifier of the voltage sensor in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage sensor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the voltage sensor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**voltage→get\_highestValue()**

**YVoltage**

**voltage→highestValue()voltage.get\_highestValue()**

---

Returns the maximal value observed for the voltage.

```
def get_highestValue( )
```

**Returns :**

a floating point number corresponding to the maximal value observed for the voltage

On failure, throws an exception or returns Y\_HIGHESTVALUE\_INVALID.

**voltage→get\_logFrequency()****YVoltage****voltage→logFrequency()voltage.get\_logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
def get_logFrequency( )
```

**Returns :**

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns Y\_LOGFREQUENCY\_INVALID.

**voltage→get\_logicalName()**

**YVoltage**

**voltage→logicalName()voltage.get\_logicalName()**

---

Returns the logical name of the voltage sensor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the voltage sensor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**voltage→get\_lowestValue()****YVoltage****voltage→lowestValue()voltage.get\_lowestValue()**

Returns the minimal value observed for the voltage.

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the voltage

On failure, throws an exception or returns Y\_LOWESTVALUE\_INVALID.

**voltage→get\_module()****YVoltage****voltage→module()voltage.get\_module()**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

---

<b>voltage→get_recordedData()</b>	<b>YVoltage</b>
<b>voltage→recordedData()voltage.get_recordedData()</b>	

---

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
def get_recordedData( startTime, endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

#### Parameters :

**startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

**endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

#### Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

**voltage→get\_reportFrequency()**  
**voltage→reportFrequency()**  
**voltage.get\_reportFrequency()**

**YVoltage**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
def get_reportFrequency( )
```

**Returns :**

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns Y\_REPORTFREQUENCY\_INVALID.

**voltage→get\_resolution()**

**YVoltage**

**voltage→resolution()voltage.get\_resolution()**

Returns the resolution of the measured values.

```
def get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

**Returns :**

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns Y\_RESOLUTION\_INVALID.

**voltage→get\_unit()**

**YVoltage**

**voltage→unit()voltage.get\_unit()**

---

Returns the measuring unit for the voltage.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**voltage→get(userData)****YVoltage****voltage→userData()voltage.get(userData())**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**voltage→isOnline()voltage.isOnline()****YVoltage**

Checks if the voltage sensor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the voltage sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage sensor.

**Returns :**

true if the voltage sensor can be reached, and false otherwise

**voltage→load()voltage.load()****YVoltage**

Preloads the voltage sensor cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**voltage→loadCalibrationPoints()****YVoltage****voltage.loadCalibrationPoints()**

Retrieves error correction data points previously entered using the method calibrateFromPoints.

```
def loadCalibrationPoints( rawValues, refValues)
```

**Parameters :**

**rawValues** array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

**refValues** array of floating point numbers, that will be filled by the function with the desired values for the correction points.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voltage→nextVoltage()voltage.nextVoltage()****YVoltage**

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

```
def nextVoltage( )
```

**Returns :**

a pointer to a `YVoltage` object, corresponding to a voltage sensor currently online, or a null pointer if there are no more voltage sensors to enumerate.

**voltage→registerTimedReportCallback()  
voltage.registerTimedReportCallback()****YVoltage**

Registers the callback function that is invoked on every periodic timed notification.

```
def registerTimedReportCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

**voltage→registerValueCallback()**  
**voltage.registerValueCallback()****YVoltage**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**voltage→set\_highestValue()**  
**voltage→setHighestValue()**  
**voltage.set\_highestValue()**

**YVoltage**

---

Changes the recorded maximal value observed pour the voltage.

```
def set_highestValue( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the recorded maximal value observed pour the voltage

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voltage→set\_logFrequency()**  
**voltage→setLogFrequency()**  
**voltage.set\_logFrequency()**

YVoltage

Changes the datalogger recording frequency for this function.

```
def set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the datalogger recording frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voltage→set\_logicalName()** **YVoltage**  
**voltage→setLogicalName()voltage.set\_logicalName()**

Changes the logical name of the voltage sensor.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the voltage sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

---

**voltage→set\_lowestValue()****YVoltage****voltage→setLowestValue()voltage.set\_lowestValue()**

---

Changes the recorded minimal value observed pour the voltage.

```
def set_lowestValue( newval )
```

**Parameters :**

**newval** a floating point number corresponding to the recorded minimal value observed pour the voltage

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voltage→set\_reportFrequency()**  
**voltage→setReportFrequency()**  
**voltage.set\_reportFrequency()**

**YVoltage**

Changes the timed value notification frequency for this function.

```
def set_reportFrequency( newval )
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

**Parameters :**

**newval** a string corresponding to the timed value notification frequency for this function

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voltage→set\_resolution()****YVoltage****voltage→setResolution()voltage.set\_resolution()**

Changes the resolution of the measured values.

```
def set_resolution( newval )
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

**Parameters :**

**newval** a floating point number corresponding to the resolution of the measured values

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**voltage→set(userData)**

**YVoltage**

**voltage→setUserData()|voltage.set(userData())**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.42. Voltage source function interface

Yoctopuce application programming interface allows you to control the module voltage output. You affect absolute output values or make transitions

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_vsource.js'></script>
php	require_once('yocto_vsource.php');
cpp	#include "yocto_vsource.h"
m	#import "yocto_vsource.h"
pas	uses yocto_vsource;
vb	yocto_vsource.vb
cs	yocto_vsource.cs
java	import com.yoctopuce.YoctoAPI.YVSource;
py	from yocto_vsource import *

### Global functions

#### yFindVSource(func)

Retrieves a voltage source for a given identifier.

#### yFirstVSource()

Starts the enumeration of voltage sources currently accessible.

### YVSource methods

#### vsource→describe()

Returns a short text that describes the function in the form TYPE ( NAME ) = SERIAL . FUNCTIONID.

#### vsource→get\_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

#### vsource→get\_errorMessage()

Returns the error message of the latest error with this function.

#### vsource→get\_errorType()

Returns the numerical error code of the latest error with this function.

#### vsource→get\_extPowerFailure()

Returns true if external power supply voltage is too low.

#### vsource→get\_failure()

Returns true if the module is in failure mode.

#### vsource→get\_friendlyName()

Returns a global identifier of the function in the format MODULE\_NAME . FUNCTION\_NAME.

#### vsource→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### vsource→get\_functionId()

Returns the hardware identifier of the function, without reference to the module.

#### vsource→get\_hardwareId()

Returns the unique hardware identifier of the function in the form SERIAL . FUNCTIONID.

#### vsource→get\_logicalName()

Returns the logical name of the voltage source.

#### vsource→get\_module()

Gets the YModule object for the device on which the function is located.

#### vsource→get\_module\_async(callback, context)

### 3. Reference

Gets the YModule object for the device on which the function is located (asynchronous version).

#### **vsouce→get\_overCurrent()**

Returns true if the appliance connected to the device is too greedy .

#### **vsouce→get\_overHeat()**

Returns TRUE if the module is overheating.

#### **vsouce→get\_overLoad()**

Returns true if the device is not able to maintain the requested voltage output .

#### **vsouce→get\_regulationFailure()**

Returns true if the voltage output is too high regarding the requested voltage .

#### **vsouce→get\_unit()**

Returns the measuring unit for the voltage.

#### **vsouce→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set(userData).

#### **vsouce→get\_voltage()**

Returns the voltage output command (mV)

#### **vsouce→isOnline()**

Checks if the function is currently reachable, without raising any error.

#### **vsouce→isOnline\_async(callback, context)**

Checks if the function is currently reachable, without raising any error (asynchronous version).

#### **vsouce→load(msValidity)**

Preloads the function cache with a specified validity duration.

#### **vsouce→load\_async(msValidity, callback, context)**

Preloads the function cache with a specified validity duration (asynchronous version).

#### **vsouce→nextVSource()**

Continues the enumeration of voltage sources started using yFirstVSource( ).

#### **vsouce→pulse(voltage, ms\_duration)**

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

#### **vsouce→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

#### **vsouce→set\_logicalName(newval)**

Changes the logical name of the voltage source.

#### **vsouce→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

#### **vsouce→set\_voltage(newval)**

Tunes the device output voltage (milliVolts).

#### **vsouce→voltageMove(target, ms\_duration)**

Performs a smooth move at constant speed toward a given value.

#### **vsouce→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**yFindVSource() —****YVSource****YVSource.FindVSource()YVSource.FindVSource()**

Retrieves a voltage source for a given identifier.

```
def FindVSource( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage source is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVSource.isOnline()` to test if the voltage source is indeed online at a given time. In case of ambiguity when looking for a voltage source by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**func** a string that uniquely characterizes the voltage source

**Returns :**

a `YVSource` object allowing you to drive the voltage source.

**yFirstVSource()** —

**YVSource**

**YVSource.FirstVSource()YVSource.FirstVSource()**

---

Starts the enumeration of voltage sources currently accessible.

```
def FirstVSource( )
```

Use the method `YVSource.nextVSource( )` to iterate on next voltage sources.

**Returns :**

a pointer to a `YVSource` object, corresponding to the first voltage source currently online, or a null pointer if there are none.

**vsource→get\_advertisedValue()**  
**vsource→advertisedValue()**  
**vsource.get\_advertisedValue()**

**YVSource**

Returns the current value of the voltage source (no more than 6 characters).

**def get\_advertisedValue( )**

**vsource→get\_advertisedValue()**  
**vsource→advertisedValue()vsource.get\_advertisedValue()**

Returns the current value of the voltage source (no more than 6 characters).

<b>js</b>	function <b>get_advertisedValue( )</b>
<b>php</b>	function <b>get_advertisedValue( )</b>
<b>cpp</b>	string <b>get_advertisedValue( )</b>
<b>m</b>	-(NSString*) <b>advertisedValue</b>
<b>pas</b>	function <b>get_advertisedValue( )</b> : string
<b>vb</b>	function <b>get_advertisedValue( )</b> As String
<b>cs</b>	string <b>get_advertisedValue( )</b>
<b>java</b>	String <b>get_advertisedValue( )</b>
<b>py</b>	def <b>get_advertisedValue( )</b>
<b>cmd</b>	YVSource <b>target get_advertisedValue</b>

**Returns :**

a string corresponding to the current value of the voltage source (no more than 6 characters)

On failure, throws an exception or returns **Y\_ADVERTISEDVALUE\_INVALID**.

**vsouce→get\_errorMessage()**  
**vsouce→errorMessage()**  
**vsouce.get\_errorMessage()**

---

**YVSource**

Returns the error message of the latest error with this function.

**def get\_errorMessage( )**

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occured while using this function object

**vsource→get\_errorType()****YVSource****vsource→errorType()vsource.get\_errorType()**

Returns the numerical error code of the latest error with this function.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using this function object

**vsouce→get\_extPowerFailure()**  
**vsouce→extPowerFailure()**  
**vsouce.get\_extPowerFailure()**

YVSource

Returns true if external power supply voltage is too low.

**def get\_extPowerFailure( )**

**vsouce→get\_extPowerFailure()**  
**vsouce→extPowerFailure()vsouce.get\_extPowerFailure()**

Returns true if external power supply voltage is too low.

**js** function get\_extPowerFailure( )  
**php** function get\_extPowerFailure( )  
**cpp** Y\_EXTPOWERFAILURE\_enum get\_extPowerFailure( )  
**m** -(Y\_EXTPOWERFAILURE\_enum) extPowerFailure  
**pas** function get\_extPowerFailure( ): Integer  
**vb** function get\_extPowerFailure( ) As Integer  
**cs** int get\_extPowerFailure( )  
**java** int get\_extPowerFailure( )  
**py** def get\_extPowerFailure( )  
**cmd** YVSource target get\_extPowerFailure

**Returns :**

either Y\_EXTPOWERFAILURE\_FALSE or Y\_EXTPOWERFAILURE\_TRUE, according to true if external power supply voltage is too low

On failure, throws an exception or returns Y\_EXTPOWERFAILURE\_INVALID.

**vsource→get\_failure()**  
**vsource→failure()vsource.get\_failure()**

YVSource

Returns true if the module is in failure mode.

```
def get_failure( )
```

**vsource→get\_failure()**  
**vsource→failure()vsource.get\_failure()**

Returns true if the module is in failure mode.

```
js function get_failure( )
php function get_failure( )
cpp Y_FAILURE_enum get_failure( )
m -(Y_FAILURE_enum) failure
pas function get_failure( ): Integer
vb function get_failure( ) As Integer
cs int get_failure( )
java int get_failure( )
py def get_failure( )
cmd YVSource target get_failure
```

More information can be obtained by testing get\_overheat, get\_overcurrent etc... When a error condition is met, the output voltage is set to zero and cannot be changed until the reset() function is called.

**Returns :**

either Y\_FAILURE\_FALSE or Y\_FAILURE\_TRUE, according to true if the module is in failure mode

On failure, throws an exception or returns Y\_FAILURE\_INVALID.

---

<b>vsources-&gt;get_functionDescriptor()</b>	<b>YVSource</b>
<b>vsources-&gt;functionDescriptor()</b>	
<b>vsources.get_vsourcDescriptor()</b>	

---

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**vsource→get\_logicalName()****YVSource****vsource→logicalName()vsource.get\_logicalName()**

Returns the logical name of the voltage source.

```
def get_logicalName( )
```

**vsource→get\_logicalName()****vsource→logicalName()vsource.get\_logicalName()**

Returns the logical name of the voltage source.

```
js function get_logicalName( )
php function get_logicalName( )
cpp string get_logicalName( )
m -(NSString*) logicalName
pas function get_logicalName( ): string
vb function get_logicalName( ) As String
cs string get_logicalName( )
java String get_logicalName( )
py def get_logicalName( )
cmd YVSource target get_logicalName
```

**Returns :**

a string corresponding to the logical name of the voltage source

On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**vsouce→get\_module()**

**YVSource**

**vsouce→module()vsouce.get\_module()**

---

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**vsource→get\_overCurrent()****YVSource****vsource→overCurrent()vsource.get\_overCurrent()**

Returns true if the appliance connected to the device is too greedy .

```
def get_overCurrent( )
```

**vsource→get\_overCurrent()****vsource→overCurrent()vsource.get\_overCurrent()**

Returns true if the appliance connected to the device is too greedy .

```
js function get_overCurrent( )
php function get_overCurrent( )
cpp Y_OVERCURRENT_enum get_overCurrent( )
m -(Y_OVERCURRENT_enum) overCurrent
pas function get_overCurrent( ): Integer
vb function get_overCurrent( ) As Integer
cs int get_overCurrent( )
java int get_overCurrent( )
py def get_overCurrent( )
cmd YVSource target get_overCurrent
```

**Returns :**

either Y\_OVERCURRENT\_FALSE or Y\_OVERCURRENT\_TRUE, according to true if the appliance connected to the device is too greedy

On failure, throws an exception or returns Y\_OVERCURRENT\_INVALID.

**vsouce→get\_overHeat()** YVSource  
**vsouce→overHeat()vsouce.get\_overHeat()**

---

Returns TRUE if the module is overheating.

```
def get_overHeat( )
```

**vsouce→get\_overHeat()**  
**vsouce→overHeat()vsouce.get\_overHeat()**

---

Returns TRUE if the module is overheating.

```
js   function get_overHeat( )
php  function get_overHeat( )
cpp  Y_OVERHEAT_enum get_overHeat( )
m    -(Y_OVERHEAT_enum) overHeat
pas   function get_overHeat( ): Integer
vb    function get_overHeat( ) As Integer
cs    int get_overHeat( )
java  int get_overHeat( )
py    def get_overHeat( )
cmd   YVSource target get_overHeat
```

**Returns :**

either Y\_OVERHEAT\_FALSE or Y\_OVERHEAT\_TRUE, according to TRUE if the module is overheating

On failure, throws an exception or returns Y\_OVERHEAT\_INVALID.

**vsource→get\_overLoad()****YVSource****vsource→overLoad()vsource.get\_overLoad()**

Returns true if the device is not able to maintain the requested voltage output .

```
def get_overLoad( )
```

**vsource→get\_overLoad()****vsource→overLoad()vsource.get\_overLoad()**

Returns true if the device is not able to maintain the requested voltage output .

```
js function get_overLoad( )
php function get_overLoad( )
cpp Y_OVERLOAD_enum get_overLoad( )
m -(Y_OVERLOAD_enum) overLoad
pas function get_overLoad( ): Integer
vb function get_overLoad( ) As Integer
cs int get_overLoad( )
java int get_overLoad( )
py def get_overLoad( )
cmd YVSource target get_overLoad
```

**Returns :**

either Y\_OVERLOAD\_FALSE or Y\_OVERLOAD\_TRUE, according to true if the device is not able to maintain the requested voltage output

On failure, throws an exception or returns Y\_OVERLOAD\_INVALID.

---

<b>vsOURCE→get_regulationFailure()</b>	<b>YVSource</b>
<b>vsOURCE→regulationFailure()</b>	
<b>vsOURCE.get_regulationFailure()</b>	

---

Returns true if the voltage output is too high regarding the requested voltage .

```
def get_regulationFailure( )
```

---

<b>vsOURCE→get_regulationFailure()</b>
<b>vsOURCE→regulationFailure()vsOURCE.get_regulationFailure()</b>

---

Returns true if the voltage output is too high regarding the requested voltage .

---

<b>js</b>	function get_regulationFailure( )
<b>php</b>	function get_regulationFailure( )
<b>cpp</b>	Y_REGULATIONFAILURE_enum get_regulationFailure( )
<b>m</b>	-(Y_REGULATIONFAILURE_enum) regulationFailure
<b>pas</b>	function get_regulationFailure( ): Integer
<b>vb</b>	function get_regulationFailure( ) As Integer
<b>cs</b>	int get_regulationFailure( )
<b>java</b>	int get_regulationFailure( )
<b>py</b>	def get_regulationFailure( )
<b>cmd</b>	YVSource target get_regulationFailure

---

**Returns :**

either Y\_REGULATIONFAILURE\_FALSE or Y\_REGULATIONFAILURE\_TRUE, according to true if the voltage output is too high regarding the requested voltage

On failure, throws an exception or returns Y\_REGULATIONFAILURE\_INVALID.

**vsource→get\_unit()****YVSource****vsource→unit()vsource.get\_unit()**

Returns the measuring unit for the voltage.

```
def get_unit( )
```

**vsource→get\_unit()****vsource→unit()vsource.get\_unit()**

Returns the measuring unit for the voltage.

```
js function get_unit( )
php function get_unit( )
cpp string get_unit( )
m -(NSString*) unit
pas function get_unit( ): string
vb function get_unit( ) As String
cs string get_unit( )
java String get_unit( )
py def get_unit( )
cmd YVSource target get_unit
```

**Returns :**

a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns Y\_UNIT\_INVALID.

**vsouce→get(userData)**

**YVSource**

**vsouce→userData()vsouce.get(userData)**

---

Returns the value of the userData attribute, as previously stored using method set(userData).

**def get(userData)**

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**vsource→get\_voltage()****YVSource****vsource→voltage()vsource.get\_voltage()**

Returns the voltage output command (mV)

```
def get_voltage( )
```

**Returns :**

an integer corresponding to the voltage output command (mV)

On failure, throws an exception or returns Y\_VOLTAGE\_INVALID.

**vsource→isOnline()vsource.isOnline()****YVSource**

Checks if the function is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

**Returns :**

`true` if the function can be reached, and `false` otherwise

**vsource→load()vsource.load()****YVSource**

Preloads the function cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

## **vsource→nextVSource()vsource.nextVSource()**

**YVSource**

Continues the enumeration of voltage sources started using `yFirstVSource()`.

```
def nextVSource( )
```

**Returns :**

a pointer to a `YVSource` object, corresponding to a voltage source currently online, or a `null` pointer if there are no more voltage sources to enumerate.

**vsource→pulse()****YVSource**

Sets device output to a specific voltage, for a specified duration, then brings it automatically to 0V.

```
def pulse( voltage, ms_duration)
```

**vsource→pulse()**

Sets device output to a specific voltage, for a specified duration, then brings it automatically to 0V.

<b>js</b>	function pulse( voltage, ms_duration)
<b>php</b>	function pulse( \$voltage, \$ms_duration)
<b>cpp</b>	int pulse( int voltage, int ms_duration)
<b>m</b>	-(int) pulse : (int) voltage : (int) ms_duration
<b>pas</b>	function pulse( voltage: integer, ms_duration: integer): integer
<b>vb</b>	function pulse( ByVal voltage As Integer, ByVal ms_duration As Integer) As Integer
<b>cs</b>	int pulse( int voltage, int ms_duration)
<b>java</b>	int pulse( int voltage, int ms_duration)
<b>py</b>	def pulse( voltage, ms_duration)
<b>cmd</b>	YVSource target pulse voltage ms_duration

**Parameters :**

**voltage**      pulse voltage, in millivolts

**ms\_duration** pulse duration, in milliseconds

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**vsouce→registerValueCallback()  
vsouce.registerValueCallback()****YVSource**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

**vsouce→registerValueCallback()vsouce.registerValueCallback()**

Registers the callback function that is invoked on every change of advertised value.

```
js function registerValueCallback( callback)
php function registerValueCallback( $callback)
cpp void registerValueCallback( YDisplayUpdateCallback callback)
pas procedure registerValueCallback( callback: TGenericUpdateCallback)
vb procedure registerValueCallback( ByVal callback As GenericUpdateCallback)
cs void registerValueCallback( UpdateCallback callback)
java void registerValueCallback( UpdateCallback callback)
py def registerValueCallback( callback)
m -(void) registerValueCallback : (YFunctionUpdateCallback) callback
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**vsource→set\_logicalName()**  
**vsource→setLogicalName()**  
**vsource.set\_logicalName()**

**YVSource**

Changes the logical name of the voltage source.

def **set\_logicalName( newval)**

**vsource→set\_logicalName()**  
**vsource→setLogicalName()vsource.set\_logicalName()**

Changes the logical name of the voltage source.

js	function <b>set_logicalName( newval)</b>
php	function <b>set_logicalName( \$newval)</b>
cpp	int <b>set_logicalName( const string&amp; newval)</b>
m	-(int) <b>setLogicalName : (NSString*) newval</b>
pas	function <b>set_logicalName( newval: string): integer</b>
vb	function <b>set_logicalName( ByVal newval As String) As Integer</b>
cs	int <b>set_logicalName( string newval)</b>
java	int <b>set_logicalName( String newval)</b>
py	def <b>set_logicalName( newval)</b>
cmd	YVSource <b>target set_logicalName newval</b>

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

#### Parameters :

**newval** a string corresponding to the logical name of the voltage source

#### Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**vsource→set(userData)**

**YVSource**

**vsource→setUserData()|vsource.set(userData)**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

**vsouce→set\_voltage()****YVSource****vsouce→setVoltage()vsouce.set\_voltage()**

Tunes the device output voltage (milliVolts).

```
def set_voltage( newval)
```

**vsouce→set\_voltage()****vsouce→setVoltage()vsouce.set\_voltage()**

Tunes the device output voltage (milliVolts).

```
[js] function set_voltage( newval)
```

```
[php] function set_voltage( $newval)
```

```
[cpp] int set_voltage( int newval)
```

```
[m] -(int) setVoltage : (int) newval
```

```
[pas] function set_voltage( newval: LongInt): integer
```

```
[vb] function set_voltage( ByVal newval As Integer) As Integer
```

```
[cs] int set_voltage( int newval)
```

```
[java] int set_voltage( int newval)
```

```
[py] def set_voltage( newval)
```

```
[cmd] YVSource target set_voltage newval
```

**Parameters :**

**newval** an integer

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**vsOURCE→voltageMove()vsOURCE.voltageMove()****YVSource**

Performs a smooth move at constant speed toward a given value.

```
def voltageMove( target, ms_duration)
```

**vsOURCE→voltageMove()vsOURCE.voltageMove()**

Performs a smooth move at constant speed toward a given value.

```
js function voltageMove( target, ms_duration)
php function voltageMove( $target, $ms_duration)
cpp int voltageMove( int target, int ms_duration)
m -(int) voltageMove : (int) target : (int) ms_duration
pas function voltageMove( target: integer, ms_duration: integer): integer
vb function voltageMove( ByVal target As Integer,
                           ByVal ms_duration As Integer) As Integer
cs int voltageMove( int target, int ms_duration)
java int voltageMove( int target, int ms_duration)
py def voltageMove( target, ms_duration)
cmd YVSource target voltageMove target ms_duration
```

**Parameters :**

**target** new output value at end of transition, in milliVolts.

**ms\_duration** transition duration, in milliseconds

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## 3.43. WakeUpMonitor function interface

The WakeUpMonitor function handles globally all wake-up sources, as well as automated sleep mode.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wakeupmonitor.js'></script>
node.js	var yoctolib = require('yoctolib');
	var YWakeUpMonitor = yoctolib.YWakeUpMonitor;
php	require_once('yocto_wakeupmonitor.php');
cpp	#include "yocto_wakeupmonitor.h"
m	#import "yocto_wakeupmonitor.h"
pas	uses yocto_wakeupmonitor;
vb	yocto_wakeupmonitor.vb
cs	yocto_wakeupmonitor.cs
java	import com.yoctopuce.YoctoAPI.YWakeUpMonitor;
py	from yocto_wakeupmonitor import *

### Global functions

#### yFindWakeUpMonitor(func)

Retrieves a monitor for a given identifier.

#### yFirstWakeUpMonitor()

Starts the enumeration of monitors currently accessible.

### YWakeUpMonitor methods

#### wakeupmonitor→describe()

Returns a short text that describes unambiguously the instance of the monitor in the form  
TYPE (NAME )=SERIAL . FUNCTIONID.

#### wakeupmonitor→get\_advertisedValue()

Returns the current value of the monitor (no more than 6 characters).

#### wakeupmonitor→get\_errorMessage()

Returns the error message of the latest error with the monitor.

#### wakeupmonitor→get\_errorType()

Returns the numerical error code of the latest error with the monitor.

#### wakeupmonitor→get\_friendlyName()

Returns a global identifier of the monitor in the format MODULE\_NAME . FUNCTION\_NAME.

#### wakeupmonitor→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### wakeupmonitor→get\_functionId()

Returns the hardware identifier of the monitor, without reference to the module.

#### wakeupmonitor→get\_hardwareId()

Returns the unique hardware identifier of the monitor in the form SERIAL . FUNCTIONID.

#### wakeupmonitor→get\_logicalName()

Returns the logical name of the monitor.

#### wakeupmonitor→get\_module()

Gets the YModule object for the device on which the function is located.

#### wakeupmonitor→get\_module\_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

#### wakeupmonitor→get\_nextWakeUp()

Returns the next scheduled wake up date/time (UNIX format)
<b>wakeupmonitor→get_powerDuration()</b>
Returns the maximal wake up time (in seconds) before automatically going to sleep.
<b>wakeupmonitor→get_sleepCountdown()</b>
Returns the delay before the next sleep period.
<b>wakeupmonitor→get(userData)</b>
Returns the value of the userData attribute, as previously stored using method <code>set(userData)</code> .
<b>wakeupmonitor→get_wakeUpReason()</b>
Returns the latest wake up reason.
<b>wakeupmonitor→get_wakeUpState()</b>
Returns the current state of the monitor
<b>wakeupmonitor→isOnline()</b>
Checks if the monitor is currently reachable, without raising any error.
<b>wakeupmonitor→isOnline_async(callback, context)</b>
Checks if the monitor is currently reachable, without raising any error (asynchronous version).
<b>wakeupmonitor→load(msValidity)</b>
Preloads the monitor cache with a specified validity duration.
<b>wakeupmonitor→load_async(msValidity, callback, context)</b>
Preloads the monitor cache with a specified validity duration (asynchronous version).
<b>wakeupmonitor→nextWakeUpMonitor()</b>
Continues the enumeration of monitors started using <code>yFirstWakeUpMonitor()</code> .
<b>wakeupmonitor→registerValueCallback(callback)</b>
Registers the callback function that is invoked on every change of advertised value.
<b>wakeupmonitor→resetSleepCountDown()</b>
Resets the sleep countdown.
<b>wakeupmonitor→set_logicalName(newval)</b>
Changes the logical name of the monitor.
<b>wakeupmonitor→set_nextWakeUp(newval)</b>
Changes the days of the week when a wake up must take place.
<b>wakeupmonitor→set_powerDuration(newval)</b>
Changes the maximal wake up time (seconds) before automatically going to sleep.
<b>wakeupmonitor→set_sleepCountdown(newval)</b>
Changes the delay before the next sleep period.
<b>wakeupmonitor→set(userData)</b>
Stores a user context provided as argument in the userData attribute of the function.
<b>wakeupmonitor→sleep(secBeforeSleep)</b>
Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.
<b>wakeupmonitor→sleepFor(secUntilWakeUp, secBeforeSleep)</b>
Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.
<b>wakeupmonitor→sleepUntil(wakeUpTime, secBeforeSleep)</b>
Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.
<b>wakeupmonitor→wait_async(callback, context)</b>

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

**wakeupmonitor→wakeUp()**

Forces a wake up.

## YWakeUpMonitor.FindWakeUpMonitor() yFindWakeUpMonitor() YWakeUpMonitor.FindWakeUpMonitor()

YWakeUpMonitor

Retrieves a monitor for a given identifier.

```
def FindWakeUpMonitor( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the monitor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeUpMonitor.isOnline()` to test if the monitor is indeed online at a given time. In case of ambiguity when looking for a monitor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the monitor

### Returns :

a `YWakeUpMonitor` object allowing you to drive the monitor.

**YWakeUpMonitor.FirstWakeUpMonitor()**  
**yFirstWakeUpMonitor()**  
**YWakeUpMonitor.FirstWakeUpMonitor()**

**YWakeUpMonitor**

Starts the enumeration of monitors currently accessible.

```
def FirstWakeUpMonitor( )
```

Use the method `YWakeUpMonitor.nextWakeUpMonitor()` to iterate on next monitors.

**Returns :**

a pointer to a `YWakeUpMonitor` object, corresponding to the first monitor currently online, or a null pointer if there are none.

**wakeupmonitor→describe()**  
**wakeupmonitor.describe()****YWakeUpMonitor**

Returns a short text that describes unambiguously the instance of the monitor in the form  
TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the monitor (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**wakeupmonitor→get\_advertisedValue()**  
**wakeupmonitor→advertisedValue()**  
**wakeupmonitor.get\_advertisedValue()****YWakeUpMonitor**

Returns the current value of the monitor (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the monitor (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**wakeupmonitor→get\_errorMessage()**  
**wakeupmonitor→errorMessage()**  
**wakeupmonitor.get\_errorMessage()**

---

**YWakeUpMonitor**

Returns the error message of the latest error with the monitor.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the monitor object

**wakeupmonitor→get\_errorType()****YWakeUpMonitor****wakeupmonitor→errorType()****wakeupmonitor.get\_errorType()**

---

Returns the numerical error code of the latest error with the monitor.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the monitor object

wakeupmonitor→get\_friendlyName()  
wakeupmonitor→friendlyName()  
wakeupmonitor.get\_friendlyName()

YWakeUpMonitor

Returns a global identifier of the monitor in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the monitor if they are defined, otherwise the serial number of the module and the hardware identifier of the monitor (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the monitor using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

wakeupmonitor→get\_functionDescriptor()  
wakeupmonitor→functionDescriptor()  
wakeupmonitor.get\_functionDescriptor()

YWakeUpMonitor

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

**def get\_functionDescriptor( )**

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**wakeupmonitor→get\_functionId()**  
**wakeupmonitor→functionId()**  
**wakeupmonitor.get\_functionId()**

---

**YWakeUpMonitor**

Returns the hardware identifier of the monitor, without reference to the module.

```
def get_functionId( )
```

For example relay1

**Returns :**

a string that identifies the monitor (ex: relay1) On failure, throws an exception or returns Y\_FUNCTIONID\_INVALID.

wakeupmonitor→get\_hardwareId()  
wakeupmonitor→hardwareId()  
wakeupmonitor.get\_hardwareId()

YWakeUpMonitor

Returns the unique hardware identifier of the monitor in the form SERIAL.FUNCTIONID.

**def get\_hardwareId( )**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the monitor. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the monitor (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

wakeupmonitor→get\_logicalName()  
wakeupmonitor→logicalName()  
wakeupmonitor.get\_logicalName()

---

YWakeUpMonitor

Returns the logical name of the monitor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the monitor. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**wakeupmonitor→get\_module()**  
**wakeupmonitor→module()**  
**wakeupmonitor.get\_module()****YWakeUpMonitor**

Gets the **YModule** object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of **YModule** is not shown as online.

**Returns :**

an instance of **YModule**

wakeupmonitor→get\_nextWakeUp()  
wakeupmonitor→nextWakeUp()  
wakeupmonitor.get\_nextWakeUp()

YWakeUpMonitor

---

Returns the next scheduled wake up date/time (UNIX format)

```
def get_nextWakeUp( )
```

**Returns :**

an integer corresponding to the next scheduled wake up date/time (UNIX format)

On failure, throws an exception or returns Y\_NEXTWAKEUP\_INVALID.

wakeupmonitor→get\_powerDuration()

YWakeUpMonitor

wakeupmonitor→powerDuration()

wakeupmonitor.get\_powerDuration()

---

Returns the maximal wake up time (in seconds) before automatically going to sleep.

```
def get_powerDuration( )
```

**Returns :**

an integer corresponding to the maximal wake up time (in seconds) before automatically going to sleep

On failure, throws an exception or returns Y\_POWERDURATION\_INVALID.

wakeupmonitor→get\_sleepCountdown()  
wakeupmonitor→sleepCountdown()  
wakeupmonitor.get\_sleepCountdown()

---

YWakeUpMonitor

Returns the delay before the next sleep period.

```
def get_sleepCountdown( )
```

**Returns :**

an integer corresponding to the delay before the next sleep period

On failure, throws an exception or returns Y\_SLEEPCOUNTDOWN\_INVALID.

wakeupmonitor→get(userData)  
wakeupmonitor→userData()  
wakeupmonitor.get(userData)

YWakeUpMonitor

Returns the value of the userData attribute, as previously stored using method setUserData.

```
def get.userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

wakeupmonitor→get\_wakeUpReason()  
wakeupmonitor→wakeUpReason()  
wakeupmonitor.get\_wakeUpReason()

YWakeUpMonitor

Returns the latest wake up reason.

```
def get_wakeUpReason( )
```

**Returns :**

a value among Y\_WAKEUPREASON\_USBPOWER, Y\_WAKEUPREASON\_EXTPOWER,  
Y\_WAKEUPREASON\_ENDOFSLEEP, Y\_WAKEUPREASON\_EXTSIG1,  
Y\_WAKEUPREASON\_EXTSIG2, Y\_WAKEUPREASON\_EXTSIG3,  
Y\_WAKEUPREASON\_EXTSIG4, Y\_WAKEUPREASON\_SCHEDULE1,  
Y\_WAKEUPREASON\_SCHEDULE2, Y\_WAKEUPREASON\_SCHEDULE3,  
Y\_WAKEUPREASON\_SCHEDULE4, Y\_WAKEUPREASON\_SCHEDULE5 and  
Y\_WAKEUPREASON\_SCHEDULE6 corresponding to the latest wake up reason

On failure, throws an exception or returns Y\_WAKEUPREASON\_INVALID.

**wakeupmonitor→get\_wakeUpState()**  
**wakeupmonitor→wakeUpState()**  
**wakeupmonitor.get\_wakeUpState()**

**YWakeUpMonitor**

Returns the current state of the monitor

```
def get_wakeUpState( )
```

**Returns :**

either `Y_WAKEUPSTATE_SLEEPING` or `Y_WAKEUPSTATE_AWAKE`, according to the current state of the monitor

On failure, throws an exception or returns `Y_WAKEUPSTATE_INVALID`.

## wakeupmonitor→isOnline()wakeupmonitor.isOnline()

## YWakeUpMonitor

---

Checks if the monitor is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the monitor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the monitor.

**Returns :**

true if the monitor can be reached, and false otherwise

**wakeupmonitor→load()wakeupmonitor.load()****YWakeUpMonitor**

Preloads the monitor cache with a specified validity duration.

```
def load( msValidity )
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**wakeupmonitor→nextWakeUpMonitor()**  
**wakeupmonitor.nextWakeUpMonitor()**

---

**YWakeUpMonitor**

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

```
def nextWakeUpMonitor( )
```

**Returns :**

a pointer to a `YWakeUpMonitor` object, corresponding to a monitor currently online, or a `null` pointer if there are no more monitors to enumerate.

**wakeupmonitor→registerValueCallback()  
wakeupmonitor.registerValueCallback()****YWakeUpMonitor**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**wakeupmonitor→resetSleepCountDown()**  
**wakeupmonitor.resetSleepCountDown()**

---

**YWakeUpMonitor**

Resets the sleep countdown.

```
def resetSleepCountDown( )
```

**Returns :**

YAPI\_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→set\_logicalName()  
wakeupmonitor→setLogicalName()  
wakeupmonitor.set\_logicalName()

YWakeUpMonitor

Changes the logical name of the monitor.

```
def set_logicalName( newval)
```

You can use yCheckLogicalName( ) prior to this call to make sure that your parameter is valid. Remember to call the saveToFlash( ) method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the monitor.

**Returns :**

YAPI\_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→set\_nextWakeUp()  
wakeupmonitor→setNextWakeUp()  
wakeupmonitor.set\_nextWakeUp()

YWakeUpMonitor

Changes the days of the week when a wake up must take place.

```
def set_nextWakeUp( newval)
```

**Parameters :**

**newval** an integer corresponding to the days of the week when a wake up must take place

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**wakeupmonitor→set\_powerDuration()**  
**wakeupmonitor→setPowerDuration()**  
**wakeupmonitor.set\_powerDuration()**

**YWakeUpMonitor**

Changes the maximal wake up time (seconds) before automatically going to sleep.

```
def set_powerDuration( newval)
```

**Parameters :**

**newval** an integer corresponding to the maximal wake up time (seconds) before automatically going to sleep

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set\_sleepCountdown()  
wakeupmonitor→setSleepCountdown()  
wakeupmonitor.set\_sleepCountdown()

YWakeUpMonitor

Changes the delay before the next sleep period.

```
def set_sleepCountdown( newval )
```

**Parameters :**

**newval** an integer corresponding to the delay before the next sleep period

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→**set(userData)**  
wakeupmonitor→**setUserData()**  
**wakeupmonitor.set(userData)**

**YWakeUpMonitor**

Stores a user context provided as argument in the userData attribute of the function.

```
def setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

**wakeupmonitor→sleep()wakeupmonitor.sleep()****YWakeUpMonitor**

Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.

```
def sleep( secBeforeSleep)
```

**Parameters :**

**secBeforeSleep** number of seconds before going into sleep mode,

**Returns :**

YAPI\_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

**wakeupmonitor→sleepFor()  
wakeupmonitor.sleepFor()****YWakeUpMonitor**

Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.

```
def sleepFor( secUntilWakeUp, secBeforeSleep)
```

The count down before sleep can be canceled with resetSleepCountDown.

**Parameters :**

**secUntilWakeUp** sleep duration, in secondes

**secBeforeSleep** number of seconds before going into sleep mode

**Returns :**

YAPI\_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

**wakeupmonitor→sleepUntil()**  
**wakeupmonitor.sleepUntil()****YWakeUpMonitor**

Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.

```
def sleepUntil( wakeUpTime, secBeforeSleep)
```

The count down before sleep can be canceled with resetSleepCountDown.

**Parameters :**

**wakeUpTime**      wake-up datetime (UNIX format)  
**secBeforeSleep**    number of seconds before going into sleep mode

**Returns :**

YAPI\_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

**wakeupmonitor→wakeUp()wakeupmonitor.wakeUp()****YWakeUpMonitor**

Forces a wake up.

```
def wakeUp( )
```

## 3.44. WakeUpSchedule function interface

The WakeUpSchedule function implements a wake up condition. The wake up time is specified as a set of months and/or days and/or hours and/or minutes when the wake up should happen.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_wakeupschedule.js'></script>
nodejs var yoctolib = require('yoctolib');
var YWakeUpSchedule = yoctolib.YWakeUpSchedule;
require_once('yocto_wakeupschedule.php');
#include "yocto_wakeupschedule.h"
m #import "yocto_wakeupschedule.h"
pas uses yocto_wakeupschedule;
vb yocto_wakeupschedule.vb
cs yocto_wakeupschedule.cs
java import com.yoctopuce.YoctoAPI.YWakeUpSchedule;
py from yocto_wakeupschedule import *

```

### Global functions

#### **yFindWakeUpSchedule(func)**

Retrieves a wake up schedule for a given identifier.

#### **yFirstWakeUpSchedule()**

Starts the enumeration of wake up schedules currently accessible.

### YWakeUpSchedule methods

#### **wakeupschedule→describe()**

Returns a short text that describes unambiguously the instance of the wake up schedule in the form  
TYPE ( NAME ) = SERIAL . FUNCTIONID.

#### **wakeupschedule→get\_advertisedValue()**

Returns the current value of the wake up schedule (no more than 6 characters).

#### **wakeupschedule→get\_errorMessage()**

Returns the error message of the latest error with the wake up schedule.

#### **wakeupschedule→get\_errorType()**

Returns the numerical error code of the latest error with the wake up schedule.

#### **wakeupschedule→get\_friendlyName()**

Returns a global identifier of the wake up schedule in the format MODULE\_NAME . FUNCTION\_NAME.

#### **wakeupschedule→get\_functionDescriptor()**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### **wakeupschedule→get\_functionId()**

Returns the hardware identifier of the wake up schedule, without reference to the module.

#### **wakeupschedule→get\_hardwareId()**

Returns the unique hardware identifier of the wake up schedule in the form SERIAL . FUNCTIONID.

#### **wakeupschedule→get\_hours()**

Returns the hours scheduled for wake up.

#### **wakeupschedule→get\_logicalName()**

Returns the logical name of the wake up schedule.

#### **wakeupschedule→get\_minutes()**

Returns all the minutes of each hour that are scheduled for wake up.

#### **wakeupschedule→get\_minutesA()**

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.
<b>wakeupschedule→get_minutesB()</b>
Returns the minutes in the 30-59 interval of each hour scheduled for wake up.
<b>wakeupschedule→get_module()</b>
Gets the YModule object for the device on which the function is located.
<b>wakeupschedule→get_module_async(callback, context)</b>
Gets the YModule object for the device on which the function is located (asynchronous version).
<b>wakeupschedule→get_monthDays()</b>
Returns the days of the month scheduled for wake up.
<b>wakeupschedule→get_months()</b>
Returns the months scheduled for wake up.
<b>wakeupschedule→get_nextOccurrence()</b>
Returns the date/time (seconds) of the next wake up occurrence
<b>wakeupschedule→get_userData()</b>
Returns the value of the userData attribute, as previously stored using method set(userData).
<b>wakeupschedule→get_weekDays()</b>
Returns the days of the week scheduled for wake up.
<b>wakeupschedule→isOnline()</b>
Checks if the wake up schedule is currently reachable, without raising any error.
<b>wakeupschedule→isOnline_async(callback, context)</b>
Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).
<b>wakeupschedule→load(msValidity)</b>
Preloads the wake up schedule cache with a specified validity duration.
<b>wakeupschedule→load_async(msValidity, callback, context)</b>
Preloads the wake up schedule cache with a specified validity duration (asynchronous version).
<b>wakeupschedule→nextWakeUpSchedule()</b>
Continues the enumeration of wake up schedules started using yFirstWakeUpSchedule().
<b>wakeupschedule→registerValueCallback(callback)</b>
Registers the callback function that is invoked on every change of advertised value.
<b>wakeupschedule→set_hours(newval)</b>
Changes the hours when a wake up must take place.
<b>wakeupschedule→set_logicalName(newval)</b>
Changes the logical name of the wake up schedule.
<b>wakeupschedule→set_minutes(bitmap)</b>
Changes all the minutes where a wake up must take place.
<b>wakeupschedule→set_minutesA(newval)</b>
Changes the minutes in the 00-29 interval when a wake up must take place.
<b>wakeupschedule→set_minutesB(newval)</b>
Changes the minutes in the 30-59 interval when a wake up must take place.
<b>wakeupschedule→set_monthDays(newval)</b>
Changes the days of the month when a wake up must take place.
<b>wakeupschedule→set_months(newval)</b>
Changes the months when a wake up must take place.
<b>wakeupschedule→set_userData(data)</b>
Stores a user context provided as argument in the userData attribute of the function.

### **3. Reference**

---

#### **wakeupschedule→set\_weekDays(newval)**

Changes the days of the week when a wake up must take place.

#### **wakeupschedule→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YWakeUpSchedule.FindWakeUpSchedule() yFindWakeUpSchedule() YWakeUpSchedule.FindWakeUpSchedule()

## YWakeUpSchedule

Retrieves a wake up schedule for a given identifier.

```
def FindWakeUpSchedule( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wake up schedule is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWakeUpSchedule.isOnline()` to test if the wake up schedule is indeed online at a given time. In case of ambiguity when looking for a wake up schedule by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the wake up schedule

### Returns :

a `YWakeUpSchedule` object allowing you to drive the wake up schedule.

## **YWakeUpSchedule.FirstWakeUpSchedule()**

## **YWakeUpSchedule**

### **yFirstWakeUpSchedule()**

### **YWakeUpSchedule.FirstWakeUpSchedule()**

---

Starts the enumeration of wake up schedules currently accessible.

```
def FirstWakeUpSchedule( )
```

Use the method `YWakeUpSchedule.nextWakeUpSchedule()` to iterate on next wake up schedules.

#### **Returns :**

a pointer to a `YWakeUpSchedule` object, corresponding to the first wake up schedule currently online, or a null pointer if there are none.

**wakeupschedule→describe()**  
**wakeupschedule.describe()****YWakeUpSchedule**

Returns a short text that describes unambiguously the instance of the wake up schedule in the form  
TYPE ( NAME ) =SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the wake up schedule (ex:  
Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**wakeupschedule→get\_advertisedValue()**  
**wakeupschedule→advertisedValue()**  
**wakeupschedule.get\_advertisedValue()**

**YWakeUpSchedule**

Returns the current value of the wake up schedule (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the wake up schedule (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**wakeupschedule→getErrorMessage()**  
**wakeupschedule→errorMessage()**  
**wakeupschedule.getErrorMessage()**

**YWakeUpSchedule**

Returns the error message of the latest error with the wake up schedule.

```
def getErrorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the wake up schedule object

**wakeupschedule→get\_errorType()**  
**wakeupschedule→errorType()**  
**wakeupschedule.get\_errorType()**

**YWakeUpSchedule**

Returns the numerical error code of the latest error with the wake up schedule.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the wake up schedule object

wakeupschedule→get\_friendlyName()  
wakeupschedule→friendlyName()  
wakeupschedule.get\_friendlyName()

YWakeUpSchedule

Returns a global identifier of the wake up schedule in the format MODULE\_NAME.FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the wake up schedule if they are defined, otherwise the serial number of the module and the hardware identifier of the wake up schedule (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the wake up schedule using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**wakeupschedule→get\_functionDescriptor()**  
**wakeupschedule→functionDescriptor()**  
**wakeupschedule.get\_functionDescriptor()**

**YWakeUpSchedule**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**wakeupschedule→get\_functionId()**  
**wakeupschedule→functionId()**  
**wakeupschedule.get\_functionId()**

**YWakeUpSchedule**

Returns the hardware identifier of the wake up schedule, without reference to the module.

```
def get_functionId( )
```

For example relay1

**Returns :**

a string that identifies the wake up schedule (ex: relay1) On failure, throws an exception or returns Y\_FUNCTIONID\_INVALID.

wakeupschedule→get\_hardwareId()  
wakeupschedule→hardwareId()  
wakeupschedule.get\_hardwareId()

YWakeUpSchedule

Returns the unique hardware identifier of the wake up schedule in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wake up schedule. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the wake up schedule (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

wakeupschedule→get\_hours()  
wakeupschedule→hours()  
wakeupschedule.get\_hours()

YWakeUpSchedule

Returns the hours scheduled for wake up.

```
def get_hours( )
```

**Returns :**

an integer corresponding to the hours scheduled for wake up

On failure, throws an exception or returns Y\_HOURS\_INVALID.

**wakeupschedule→get\_logicalName()**  
**wakeupschedule→logicalName()**  
**wakeupschedule.get\_logicalName()**

---

**YWakeUpSchedule**

Returns the logical name of the wake up schedule.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the wake up schedule. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

wakeupschedule→get\_minutes()  
wakeupschedule→minutes()  
wakeupschedule.get\_minutes()

YWakeUpSchedule

---

Returns all the minutes of each hour that are scheduled for wake up.

```
def get_minutes( )
```

**wakeupschedule→get\_minutesA()**  
**wakeupschedule→minutesA()**  
**wakeupschedule.get\_minutesA()**

**YWakeUpSchedule**

---

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

```
def get_minutesA( )
```

**Returns :**

an integer corresponding to the minutes in the 00-29 interval of each hour scheduled for wake up

On failure, throws an exception or returns Y\_MINUTESA\_INVALID.

wakeupschedule→get\_minutesB()  
wakeupschedule→minutesB()  
wakeupschedule.get\_minutesB()

YWakeUpSchedule

Returns the minutes in the 30-59 interval of each hour scheduled for wake up.

```
def get_minutesB( )
```

**Returns :**

an integer corresponding to the minutes in the 30-59 interval of each hour scheduled for wake up

On failure, throws an exception or returns Y\_MINUTESB\_INVALID.

**wakeupschedule→get\_module()**  
**wakeupschedule→module()**  
**wakeupschedule.get\_module()**

**YWakeUpSchedule**

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

wakeupschedule→get\_monthDays()

YWakeUpSchedule

wakeupschedule→monthDays()

wakeupschedule.get\_monthDays()

---

Returns the days of the month scheduled for wake up.

```
def get_monthDays( )
```

**Returns :**

an integer corresponding to the days of the month scheduled for wake up

On failure, throws an exception or returns Y\_MONTHDAYS\_INVALID.

wakeupschedule→get\_months()

YWakeUpSchedule

wakeupschedule→months()

wakeupschedule.get\_months()

---

Returns the months scheduled for wake up.

```
def get_months( )
```

**Returns :**

an integer corresponding to the months scheduled for wake up

On failure, throws an exception or returns Y\_MONTHS\_INVALID.

wakeupschedule→get\_nextOccurence()  
wakeupschedule→nextOccurence()  
wakeupschedule.get\_nextOccurence()

YWakeUpSchedule

Returns the date/time (seconds) of the next wake up occurence

```
def get_nextOccurence( )
```

**Returns :**

an integer corresponding to the date/time (seconds) of the next wake up occurence

On failure, throws an exception or returns Y\_NEXTOCCURENCE\_INVALID.

**wakeupschedule→get(userData)**

**YWakeUpSchedule**

**wakeupschedule→userData()**

**wakeupschedule.get(userData)**

---

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

wakeupschedule→get\_weekDays()  
wakeupschedule→weekDays()  
wakeupschedule.get\_weekDays()

YWakeUpSchedule

Returns the days of the week scheduled for wake up.

```
def get_weekDays( )
```

**Returns :**

an integer corresponding to the days of the week scheduled for wake up

On failure, throws an exception or returns Y\_WEEKDAYS\_INVALID.

**wakeupschedule→isOnline()**  
**wakeupschedule.isOnline()****YWakeUpSchedule**

Checks if the wake up schedule is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the wake up schedule in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wake up schedule.

**Returns :**

true if the wake up schedule can be reached, and false otherwise

**wakeupschedule→load()wakeupschedule.load()****YWakeUpSchedule**

Preloads the wake up schedule cache with a specified validity duration.

```
def load( msValidity )
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**wakeupschedule→nextWakeUpSchedule()**  
**wakeupschedule.nextWakeUpSchedule()**

**YWakeUpSchedule**

Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

```
def nextWakeUpSchedule( )
```

**Returns :**

a pointer to a `YWakeUpSchedule` object, corresponding to a wake up schedule currently online, or a null pointer if there are no more wake up schedules to enumerate.

**wakeupschedule→registerValueCallback()  
wakeupschedule.registerValueCallback()****YWakeUpSchedule**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

wakeupschedule→set\_hours()  
wakeupschedule→setHours()  
wakeupschedule.set\_hours()

YWakeUpSchedule

---

Changes the hours when a wake up must take place.

```
def set_hours( newval)
```

**Parameters :**

**newval** an integer corresponding to the hours when a wake up must take place

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set\_logicalName()  
wakeupschedule→setLogicalName()  
wakeupschedule.set\_logicalName()

YWakeUpSchedule

Changes the logical name of the wake up schedule.

```
def set_logicalName( newval)
```

You can use yCheckLogicalName( ) prior to this call to make sure that your parameter is valid. Remember to call the saveToFlash( ) method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the wake up schedule.

**Returns :**

YAPI\_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

**wakeupschedule→set\_minutes()**  
**wakeupschedule→setMinutes()**  
**wakeupschedule.set\_minutes()**

**YWakeUpSchedule**

---

Changes all the minutes where a wake up must take place.

```
def set_minutes( bitmap)
```

**Parameters :**

**bitmap** Minutes 00-59 of each hour scheduled for wake up.

**Returns :**

YAPI\_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupschedule→set\_minutesA()  
wakeupschedule→setMinutesA()  
wakeupschedule.set\_minutesA()

YWakeUpSchedule

Changes the minutes in the 00-29 interval when a wake up must take place.

```
def set_minutesA( newval)
```

**Parameters :**

**newval** an integer corresponding to the minutes in the 00-29 interval when a wake up must take place

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**wakeupschedule→set\_minutesB()**  
**wakeupschedule→setMinutesB()**  
**wakeupschedule.set\_minutesB()**

**YWakeUpSchedule**

Changes the minutes in the 30-59 interval when a wake up must take place.

```
def set_minutesB( newval)
```

**Parameters :**

**newval** an integer corresponding to the minutes in the 30-59 interval when a wake up must take place

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set\_monthDays()  
wakeupschedule→setMonthDays()  
wakeupschedule.set\_monthDays()

YWakeUpSchedule

Changes the days of the month when a wake up must take place.

```
def set_monthDays( newval)
```

**Parameters :**

**newval** an integer corresponding to the days of the month when a wake up must take place

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set\_months()  
wakeupschedule→setMonths()  
wakeupschedule.set\_months()

YWakeUpSchedule

Changes the months when a wake up must take place.

```
def set_months( newval )
```

**Parameters :**

**newval** an integer corresponding to the months when a wake up must take place

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**wakeupschedule→set(userData)**  
**wakeupschedule→setUserData()**  
**wakeupschedule.set(userData)**

**YWakeUpSchedule**

Stores a user context provided as argument in the userData attribute of the function.

```
def setUserData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

wakeupschedule→set\_weekDays()  
wakeupschedule→setWeekDays()  
wakeupschedule.set\_weekDays()

YWakeUpSchedule

Changes the days of the week when a wake up must take place.

```
def set_weekDays( newval)
```

**Parameters :**

**newval** an integer corresponding to the days of the week when a wake up must take place

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## 3.45. Watchdog function interface

The watchdog function works like a relay and can cause a brief power cut to an appliance after a preset delay to force this appliance to reset. The Watchdog must be called from time to time to reset the timer and prevent the appliance reset. The watchdog can be driven directly with *pulse* and *delayedpulse* methods to switch off an appliance for a given duration.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_watchdog.js'></script>
node.js	var yoctolib = require('yoctolib');
	var YWatchdog = yoctolib.YWatchdog;
php	require_once('yocto_watchdog.php');
cpp	#include "yocto_watchdog.h"
m	#import "yocto_watchdog.h"
pas	uses yocto_watchdog;
vb	yocto_watchdog.vb
cs	yocto_watchdog.cs
java	import com.yoctopuce.YoctoAPI.YWatchdog;
py	from yocto_watchdog import *

### Global functions

#### yFindWatchdog(func)

Retrieves a watchdog for a given identifier.

#### yFirstWatchdog()

Starts the enumeration of watchdog currently accessible.

### YWatchdog methods

#### watchdog->delayedPulse(ms\_delay, ms\_duration)

Schedules a pulse.

#### watchdog->describe()

Returns a short text that describes unambiguously the instance of the watchdog in the form  
TYPE (NAME) = SERIAL.FUNCTIONID.

#### watchdog->get\_advertisedValue()

Returns the current value of the watchdog (no more than 6 characters).

#### watchdog->get\_autoStart()

Returns the watchdog running state at module power on.

#### watchdog->get\_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call). When there is no scheduled pulse, returns zero.

#### watchdog->get\_errorMessage()

Returns the error message of the latest error with the watchdog.

#### watchdog->get\_errorType()

Returns the numerical error code of the latest error with the watchdog.

#### watchdog->get\_friendlyName()

Returns a global identifier of the watchdog in the format MODULE\_NAME . FUNCTION\_NAME.

#### watchdog->get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### watchdog->get\_functionId()

Returns the hardware identifier of the watchdog, without reference to the module.

<b>watchdog→get_hardwareId()</b>	Returns the unique hardware identifier of the watchdog in the form SERIAL . FUNCTIONID.
<b>watchdog→get_logicalName()</b>	Returns the logical name of the watchdog.
<b>watchdog→get_maxTimeOnStateA()</b>	Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.
<b>watchdog→get_maxTimeOnStateB()</b>	Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.
<b>watchdog→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>watchdog→get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>watchdog→get_output()</b>	Returns the output state of the watchdog, when used as a simple switch (single throw).
<b>watchdog→get_pulseTimer()</b>	Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.
<b>watchdog→get_running()</b>	Returns the watchdog running state.
<b>watchdog→get_state()</b>	Returns the state of the watchdog (A for the idle position, B for the active position).
<b>watchdog→get_stateAtPowerOn()</b>	Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).
<b>watchdog→get_triggerDelay()</b>	Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.
<b>watchdog→get_triggerDuration()</b>	Returns the duration of resets caused by the watchdog, in milliseconds.
<b>watchdog→get_userData()</b>	Returns the value of the userData attribute, as previously stored using method set(userData).
<b>watchdog→isOnline()</b>	Checks if the watchdog is currently reachable, without raising any error.
<b>watchdog→isOnline_async(callback, context)</b>	Checks if the watchdog is currently reachable, without raising any error (asynchronous version).
<b>watchdog→load(msValidity)</b>	Preloads the watchdog cache with a specified validity duration.
<b>watchdog→load_async(msValidity, callback, context)</b>	Preloads the watchdog cache with a specified validity duration (asynchronous version).
<b>watchdog→nextWatchdog()</b>	Continues the enumeration of watchdog started using yFirstWatchdog( ).
<b>watchdog→pulse(ms_duration)</b>	Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).
<b>watchdog→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.

**watchdog->resetWatchdog()**

Resets the watchdog.

**watchdog->set\_autoStart(newval)**

Changes the watchdog running state at module power on.

**watchdog->set\_logicalName(newval)**

Changes the logical name of the watchdog.

**watchdog->set\_maxTimeOnStateA(newval)**

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

**watchdog->set\_maxTimeOnStateB(newval)**

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

**watchdog->set\_output(newval)**

Changes the output state of the watchdog, when used as a simple switch (single throw).

**watchdog->set\_running(newval)**

Changes the running state of the watchdog.

**watchdog->set\_state(newval)**

Changes the state of the watchdog (A for the idle position, B for the active position).

**watchdog->set\_stateAtPowerOn(newval)**

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

**watchdog->set\_triggerDelay(newval)**

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

**watchdog->set\_triggerDuration(newval)**

Changes the duration of resets caused by the watchdog, in milliseconds.

**watchdog->set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**watchdog->wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YWatchdog.FindWatchdog() yFindWatchdog()YWatchdog.FindWatchdog()

Retrieves a watchdog for a given identifier.

```
def FindWatchdog( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the watchdog is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWatchdog.isOnline()` to test if the watchdog is indeed online at a given time. In case of ambiguity when looking for a watchdog by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

### Parameters :

`func` a string that uniquely characterizes the watchdog

### Returns :

a `YWatchdog` object allowing you to drive the watchdog.

**YWatchdog.FirstWatchdog()****YWatchdog****yFirstWatchdog()YWatchdog.FirstWatchdog()**

Starts the enumeration of watchdog currently accessible.

```
def FirstWatchdog( )
```

Use the method `YWatchdog.nextWatchdog()` to iterate on next watchdog.

**Returns :**

a pointer to a `YWatchdog` object, corresponding to the first watchdog currently online, or a `null` pointer if there are none.

**watchdog→delayedPulse()|watchdog.delayedPulse()****YWatchdog**

Schedules a pulse.

```
def delayedPulse( ms_delay, ms_duration)
```

**Parameters :**

**ms\_delay** waiting time before the pulse, in millisecondes

**ms\_duration** pulse duration, in millisecondes

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**watchdog→describe()watchdog.describe()****YWatchdog**

Returns a short text that describes unambiguously the instance of the watchdog in the form TYPE ( NAME )=SERIAL.FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the watchdog (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**watchdog→get\_advertisedValue()**  
**watchdog→advertisedValue()**  
**watchdog.get\_advertisedValue()**

**YWatchdog**

Returns the current value of the watchdog (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the watchdog (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**watchdog→get\_autoStart()****YWatchdog****watchdog→autoStart()watchdog.get\_autoStart()**

Returns the watchdog runing state at module power on.

```
def get_autoStart( )
```

**Returns :**

either Y\_AUTOSTART\_OFF or Y\_AUTOSTART\_ON, according to the watchdog runing state at module power on

On failure, throws an exception or returns Y\_AUTOSTART\_INVALID.

**watchdog→get\_countdown()****YWatchdog****watchdog→countdown()watchdog.get\_countdown()**

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

```
def get_countdown( )
```

**Returns :**

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns Y\_COUNTDOWN\_INVALID.

---

**watchdog→get\_errorMessage()**  
**watchdog→errorMessage()**  
**watchdog.get\_errorMessage()****YWatchdog**

Returns the error message of the latest error with the watchdog.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the watchdog object

**watchdog→get\_errorType()**

**YWatchdog**

**watchdog→errorType()watchdog.get\_errorType()**

---

Returns the numerical error code of the latest error with the watchdog.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the watchdog object

**watchdog→get\_friendlyName()**  
**watchdog→friendlyName()**  
**watchdog.get\_friendlyName()****YWatchdog**

Returns a global identifier of the watchdog in the format MODULE\_NAME . FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the watchdog if they are defined, otherwise the serial number of the module and the hardware identifier of the watchdog (for exemple: MyCustomName . relay1)

**Returns :**

a string that uniquely identifies the watchdog using logical names (ex: MyCustomName . relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**watchdog→get\_functionDescriptor()**  
**watchdog→functionDescriptor()**  
**watchdog.get\_functionDescriptor()**

**YWatchdog**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

**watchdog→get\_functionId()****YWatchdog****watchdog→functionId()watchdog.get\_functionId()**

Returns the hardware identifier of the watchdog, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the watchdog (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**watchdog→get\_hardwareId()**

**YWatchdog**

**watchdog→hardwareId()watchdog.get\_hardwareId()**

---

Returns the unique hardware identifier of the watchdog in the form SERIAL.FUNCTIONID.

```
def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the watchdog. (for example RELAYLO1-123456.relay1)

**Returns :**

a string that uniquely identifies the watchdog (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**watchdog→get\_logicalName()  
watchdog→logicalName()  
watchdog.get\_logicalName()****YWatchdog**

Returns the logical name of the watchdog.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the watchdog. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

**watchdog→get\_maxTimeOnStateA()**  
**watchdog→maxTimeOnStateA()**  
**watchdog.get\_maxTimeOnStateA()**

**YWatchdog**

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

```
def get_maxTimeOnStateA( )
```

Zero means no maximum time.

**Returns :**

an integer

On failure, throws an exception or returns Y\_MAXTIMEONSTATEA\_INVALID.

**watchdog→get\_maxTimeOnStateB()**  
**watchdog→maxTimeOnStateB()**  
**watchdog.get\_maxTimeOnStateB()****YWatchdog**

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

```
def get_maxTimeOnStateB( )
```

Zero means no maximum time.

**Returns :**

an integer

On failure, throws an exception or returns Y\_MAXTIMEONSTATEB\_INVALID.

**watchdog→get\_module()**

**YWatchdog**

**watchdog→module()watchdog.get\_module()**

---

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

**Returns :**

an instance of `YModule`

**watchdog→get\_output()****YWatchdog****watchdog→output()watchdog.get\_output()**

Returns the output state of the watchdog, when used as a simple switch (single throw).

```
def get_output( )
```

**Returns :**

either Y\_OUTPUT\_OFF or Y\_OUTPUT\_ON, according to the output state of the watchdog, when used as a simple switch (single throw)

On failure, throws an exception or returns Y\_OUTPUT\_INVALID.

**watchdog→get\_pulseTimer()** **YWatchdog**  
**watchdog→pulseTimer()watchdog.get\_pulseTimer()**

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

```
def get_pulseTimer( )
```

When there is no ongoing pulse, returns zero.

**Returns :**

an integer corresponding to the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns Y\_PULSE\_TIMER\_INVALID.

**watchdog→get\_running()****YWatchdog****watchdog→running()watchdog.get\_running()**

Returns the watchdog running state.

```
def get_running( )
```

**Returns :**

either Y\_RUNNING\_OFF or Y\_RUNNING\_ON, according to the watchdog running state

On failure, throws an exception or returns Y\_RUNNING\_INVALID.

**watchdog→get\_state()**

**YWatchdog**

**watchdog→state()watchdog.get\_state()**

---

Returns the state of the watchdog (A for the idle position, B for the active position).

```
def get_state( )
```

**Returns :**

either Y\_STATE\_A or Y\_STATE\_B, according to the state of the watchdog (A for the idle position, B for the active position)

On failure, throws an exception or returns Y\_STATE\_INVALID.

**watchdog→get\_stateAtPowerOn()**  
**watchdog→stateAtPowerOn()**  
**watchdog.get\_stateAtPowerOn()**

**YWatchdog**

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

```
def get_stateAtPowerOn( )
```

**Returns :**

a value among Y\_STATEATPOWERON\_UNCHANGED, Y\_STATEATPOWERON\_A and Y\_STATEATPOWERON\_B corresponding to the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns Y\_STATEATPOWERON\_INVALID.

**watchdog→get\_triggerDelay()**  
**watchdog→triggerDelay()**  
**watchdog.get\_triggerDelay()**

**YWatchdog**

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

```
def get_triggerDelay( )
```

**Returns :**

an integer corresponding to the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds

On failure, throws an exception or returns Y\_TRIGGERDELAY\_INVALID.

**watchdog→get\_triggerDuration()**  
**watchdog→triggerDuration()**  
**watchdog.get\_triggerDuration()****YWatchdog**

Returns the duration of resets caused by the watchdog, in milliseconds.

```
def get_triggerDuration( )
```

**Returns :**

an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

On failure, throws an exception or returns Y\_TRIGGERDURATION\_INVALID.

**watchdog→get(userData)**

**YWatchdog**

**watchdog→userData()watchdog.get(userData)**

---

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**watchdog→isOnline()watchdog.isOnline()****YWatchdog**

Checks if the watchdog is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the watchdog in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the watchdog.

**Returns :**

`true` if the watchdog can be reached, and `false` otherwise

**watchdog→load()|watchdog.load()****YWatchdog**

Preloads the watchdog cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**watchdog→nextWatchdog()  
watchdog.nextWatchdog()****YWatchdog**

Continues the enumeration of watchdog started using `yFirstWatchdog( )`.

```
def nextWatchdog( )
```

**Returns :**

a pointer to a `YWatchdog` object, corresponding to a watchdog currently online, or a `null` pointer if there are no more watchdog to enumerate.

**watchdog→pulse()watchdog.pulse()****YWatchdog**

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

```
def pulse( ms_duration )
```

**Parameters :**

**ms\_duration** pulse duration, in millisecondes

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**watchdog→registerValueCallback()**  
**watchdog.registerValueCallback()****YWatchdog**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## **watchdog→resetWatchdog() watchdog.resetWatchdog()**

---

**YWatchdog**

Resets the watchdog.

```
def resetWatchdog( )
```

When the watchdog is running, this function must be called on a regular basis to prevent the watchdog to trigger

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**watchdog→set\_autoStart()****YWatchdog****watchdog→setAutoStart()watchdog.set\_autoStart()**

Changes the watchdog runningsttae at module power on.

```
def set_autoStart( newval)
```

Remember to call the saveToFlash( ) method and then to reboot the module to apply this setting.

**Parameters :**

**newval** either Y\_AUTOSTART\_OFF or Y\_AUTOSTART\_ON, according to the watchdog runningsttae at module power on

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**watchdog→set\_logicalName()**  
**watchdog→setLogicalName()**  
**watchdog.set\_logicalName()**

**YWatchdog**

Changes the logical name of the watchdog.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a string corresponding to the logical name of the watchdog.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**watchdog→set\_maxTimeOnStateA()**  
**watchdog→setMaxTimeOnStateA()**  
**watchdog.set\_maxTimeOnStateA()**

**YWatchdog**

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

```
def set_maxTimeOnStateA( newval)
```

Use zero for no maximum time.

**Parameters :**

**newval** an integer

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**watchdog→set\_maxTimeOnStateB()**  
**watchdog→setMaxTimeOnStateB()**  
**watchdog.set\_maxTimeOnStateB()**

**YWatchdog**

Sets the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

```
def set_maxTimeOnStateB( newval )
```

Use zero for no maximum time.

**Parameters :**

**newval** an integer

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**watchdog→set\_output()****YWatchdog****watchdog→setOutput()watchdog.set\_output()**

Changes the output state of the watchdog, when used as a simple switch (single throw).

```
def set_output( newval)
```

**Parameters :**

**newval** either Y\_OUTPUT\_OFF or Y\_OUTPUT\_ON, according to the output state of the watchdog, when used as a simple switch (single throw)

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

## watchdog→set\_running()

YWatchdog

## watchdog→setRunning()watchdog.set\_running()

---

Changes the running state of the watchdog.

```
def set_running( newval)
```

### Parameters :

**newval** either Y\_RUNNING\_OFF or Y\_RUNNING\_ON, according to the running state of the watchdog

### Returns :

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**watchdog→set\_state()****YWatchdog****watchdog→setState()watchdog.set\_state()**

Changes the state of the watchdog (A for the idle position, B for the active position).

```
def set_state( newval)
```

**Parameters :**

**newval** either Y\_STATE\_A or Y\_STATE\_B, according to the state of the watchdog (A for the idle position, B for the active position)

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**watchdog→set\_stateAtPowerOn()**  
**watchdog→setStateAtPowerOn()**  
**watchdog.set\_stateAtPowerOn()**

**YWatchdog**

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

```
def set_stateAtPowerOn( newval )
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

**Parameters :**

**newval** a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B`

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**watchdog→set\_triggerDelay()**  
**watchdog→setTriggerDelay()**  
**watchdog.set\_triggerDelay()**

**YWatchdog**

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

```
def set_triggerDelay( newval)
```

**Parameters :**

**newval** an integer corresponding to the waiting delay before a reset is triggered by the watchdog, in milliseconds

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**watchdog→set\_triggerDuration()**  
**watchdog→setTriggerDuration()**  
**watchdog.set\_triggerDuration()**

**YWatchdog**

Changes the duration of resets caused by the watchdog, in milliseconds.

```
def set_triggerDuration( newval)
```

**Parameters :**

**newval** an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

**Returns :**

YAPI\_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**watchdog→set(userData())****YWatchdog****watchdog→setUserData()watchdog.set(userData())**

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## 3.46. Wireless function interface

YWireless functions provides control over wireless network parameters and status for devices that are wireless-enabled.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wireless.js'></script>
nodejs	var yoctolib = require('yoctolib');
	var YWireless = yoctolib.YWireless;
php	require_once('yocto_wireless.php');
cpp	#include "yocto_wireless.h"
m	#import "yocto_wireless.h"
pas	uses yocto_wireless;
vb	yocto_wireless.vb
cs	yocto_wireless.cs
java	import com.yoctopuce.YoctoAPI.YWireless;
py	from yocto_wireless import *

### Global functions

#### yFindWireless(func)

Retrieves a wireless lan interface for a given identifier.

#### yFirstWireless()

Starts the enumeration of wireless lan interfaces currently accessible.

### YWireless methods

#### wireless→adhocNetwork(ssid, securityKey)

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

#### wireless→describe()

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form TYPE ( NAME ) = SERIAL . FUNCTIONID.

#### wireless→get\_advertisedValue()

Returns the current value of the wireless lan interface (no more than 6 characters).

#### wireless→get\_channel()

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

#### wireless→get\_detectedWlans()

Returns a list of YWlanRecord objects that describe detected Wireless networks.

#### wireless→get\_errorMessage()

Returns the error message of the latest error with the wireless lan interface.

#### wireless→get\_errorType()

Returns the numerical error code of the latest error with the wireless lan interface.

#### wireless→get\_friendlyName()

Returns a global identifier of the wireless lan interface in the format MODULE\_NAME . FUNCTION\_NAME.

#### wireless→get\_functionDescriptor()

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

#### wireless→get\_functionId()

Returns the hardware identifier of the wireless lan interface, without reference to the module.

#### wireless→get\_hardwareId()

Returns the unique hardware identifier of the wireless lan interface in the form SERIAL . FUNCTIONID.

**wireless→get\_linkQuality()**

Returns the link quality, expressed in percent.

**wireless→get\_logicalName()**

Returns the logical name of the wireless lan interface.

**wireless→get\_message()**

Returns the latest status message from the wireless interface.

**wireless→get\_module()**

Gets the YModule object for the device on which the function is located.

**wireless→get\_module\_async(callback, context)**

Gets the YModule object for the device on which the function is located (asynchronous version).

**wireless→get\_security()**

Returns the security algorithm used by the selected wireless network.

**wireless→get\_ssid()**

Returns the wireless network name (SSID).

**wireless→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set(userData).

**wireless→isOnline()**

Checks if the wireless lan interface is currently reachable, without raising any error.

**wireless→isOnline\_async(callback, context)**

Checks if the wireless lan interface is currently reachable, without raising any error (asynchronous version).

**wireless→joinNetwork(ssid, securityKey)**

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

**wireless→load(msValidity)**

Preloads the wireless lan interface cache with a specified validity duration.

**wireless→load\_async(msValidity, callback, context)**

Preloads the wireless lan interface cache with a specified validity duration (asynchronous version).

**wireless→nextWireless()**

Continues the enumeration of wireless lan interfaces started using yFirstWireless( ).

**wireless→registerValueCallback(callback)**

Registers the callback function that is invoked on every change of advertised value.

**wireless→set\_logicalName(newval)**

Changes the logical name of the wireless lan interface.

**wireless→set\_userData(data)**

Stores a user context provided as argument in the userData attribute of the function.

**wireless→wait\_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YWireless.FindWireless() yFindWireless()YWireless.FindWireless()

YWireless

Retrieves a wireless lan interface for a given identifier.

```
def FindWireless( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the wireless lan interface is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YWireless.isOnline()` to test if the wireless lan interface is indeed online at a given time. In case of ambiguity when looking for a wireless lan interface by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

`func` a string that uniquely characterizes the wireless lan interface

**Returns :**

a `YWireless` object allowing you to drive the wireless lan interface.

**YWireless.FirstWireless()****YWireless****yFirstWireless()YWireless.FirstWireless()**

Starts the enumeration of wireless lan interfaces currently accessible.

```
def FirstWireless( )
```

Use the method `YWireless.nextWireless( )` to iterate on next wireless lan interfaces.

**Returns :**

a pointer to a `YWireless` object, corresponding to the first wireless lan interface currently online, or a null pointer if there are none.

**wireless→adhocNetwork()|wireless.adhocNetwork()****YWireless**

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

```
def adhocNetwork( ssid, securityKey)
```

If a security key is specified, the network is protected by WEP128, since WPA is not standardized for ad-hoc networks. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

**Parameters :**

**ssid** the name of the network to connect to  
**securityKey** the network key, as a character string

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**wireless→describe()wireless.describe()****YWireless**

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form TYPE ( NAME )=SERIAL . FUNCTIONID.

```
def describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**

a string that describes the wireless lan interface (ex:  
Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

**wireless→get\_advertisedValue()**  
**wireless→advertisedValue()**  
**wireless.get\_advertisedValue()**

**YWireless**

---

Returns the current value of the wireless lan interface (no more than 6 characters).

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the wireless lan interface (no more than 6 characters). On failure, throws an exception or returns Y\_ADVERTISEDVALUE\_INVALID.

**wireless→get\_channel()****YWireless****wireless→channel()wireless.get\_channel()**

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

```
def get_channel( )
```

**Returns :**

an integer corresponding to the 802.11 channel currently used, or 0 when the selected network has not been found

On failure, throws an exception or returns Y\_CHANNEL\_INVALID.

**wireless→get\_detectedWlans()**  
**wireless→detectedWlans()**  
**wireless.get\_detectedWlans()**

**YWireless**

Returns a list of YWlanRecord objects that describe detected Wireless networks.

```
def get_detectedWlans( )
```

This list is not updated when the module is already connected to an acces point (infrastructure mode). To force an update of this list, adhocNetwork( ) must be called to disconnect the module from the current network. The returned list must be unallocated by the caller.

**Returns :**

a list of YWlanRecord objects, containing the SSID, channel, link quality and the type of security of the wireless network.

On failure, throws an exception or returns an empty list.

**wireless→get\_errorMessage()**  
**wireless→errorMessage()**  
**wireless.get\_errorMessage()**

**YWireless**

---

Returns the error message of the latest error with the wireless lan interface.

```
def get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a string corresponding to the latest error message that occurred while using the wireless lan interface object

**wireless→get\_errorType()**

**YWireless**

**wireless→errorType()wireless.get\_errorType()**

---

Returns the numerical error code of the latest error with the wireless lan interface.

```
def get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**

a number corresponding to the code of the latest error that occurred while using the wireless lan interface object

**wireless→get\_friendlyName()****YWireless****wireless→friendlyName()wireless.get\_friendlyName()**

Returns a global identifier of the wireless lan interface in the format MODULE\_NAME.FUNCTION\_NAME.

```
def get_friendlyName( )
```

The returned string uses the logical names of the module and of the wireless lan interface if they are defined, otherwise the serial number of the module and the hardware identifier of the wireless lan interface (for exemple: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the wireless lan interface using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y\_FRIENDLYNAME\_INVALID.

**wireless→get\_functionDescriptor()**  
**wireless→functionDescriptor()**  
**wireless.get\_functionDescriptor()**

**YWireless**

Returns a unique identifier of type YFUN\_DESCR corresponding to the function.

```
def get_functionDescriptor( )
```

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**

an identifier of type YFUN\_DESCR. If the function has never been contacted, the returned value is Y\_FUNCTIONDESCRIPTOR\_INVALID.

---

**wireless→get\_functionId()****YWireless****wireless→functionId()wireless.get\_functionId()**

---

Returns the hardware identifier of the wireless lan interface, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the wireless lan interface (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

**wireless→get\_hwId()**

**YWireless**

**wireless→hardwareId()wireless.get\_hwId()**

---

Returns the unique hardware identifier of the wireless lan interface in the form SERIAL.FUNCTIONID.

```
def get_hwId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the wireless lan interface. (for example RELAYL01-123456.relay1)

**Returns :**

a string that uniquely identifies the wireless lan interface (ex: RELAYL01-123456.relay1) On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

**wireless→get\_linkQuality()**

**YWireless**

**wireless→linkQuality()wireless.get\_linkQuality()**

Returns the link quality, expressed in percent.

```
def get_linkQuality( )
```

**Returns :**

an integer corresponding to the link quality, expressed in percent

On failure, throws an exception or returns Y\_LINKQUALITY\_INVALID.

**wireless→get\_logicalName()** YWireless  
**wireless→logicalName()wireless.get\_logicalName()**

---

Returns the logical name of the wireless lan interface.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the wireless lan interface. On failure, throws an exception or returns Y\_LOGICALNAME\_INVALID.

---

**wireless→get\_message()****YWireless****wireless→message()wireless.get\_message()**

---

Returns the latest status message from the wireless interface.

```
def get_message( )
```

**Returns :**

a string corresponding to the latest status message from the wireless interface

On failure, throws an exception or returns Y\_MESSAGE\_INVALID.

**wireless→get\_module()**

**YWireless**

**wireless→module()wireless.get\_module()**

---

Gets the `YModule` object for the device on which the function is located.

```
def get_module( )
```

If the function cannot be located on any module, the returned instance of `YModule` is not shown as on-line.

**Returns :**

an instance of `YModule`

**wireless→get\_security()****YWireless****wireless→security()wireless.get\_security()**

Returns the security algorithm used by the selected wireless network.

```
def get_security( )
```

**Returns :**

a value among Y\_SECURITY\_UNKNOWN, Y\_SECURITY\_OPEN, Y\_SECURITY\_WEP, Y\_SECURITY\_WPA and Y\_SECURITY\_WPA2 corresponding to the security algorithm used by the selected wireless network

On failure, throws an exception or returns Y\_SECURITY\_INVALID.

**wireless→get\_ssid()**

**YWireless**

**wireless→ssid()wireless.get\_ssid()**

---

Returns the wireless network name (SSID).

```
def get_ssid( )
```

**Returns :**

a string corresponding to the wireless network name (SSID)

On failure, throws an exception or returns Y\_SSID\_INVALID.

---

**wireless→get(userData)****YWireless****wireless→userData()wireless.get(userData())**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
def get(userData):
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**

the object stored previously by the caller.

**wireless→isOnline()wireless.isOnline()****YWireless**

Checks if the wireless lan interface is currently reachable, without raising any error.

```
def isOnline( )
```

If there is a cached value for the wireless lan interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the wireless lan interface.

**Returns :**

`true` if the wireless lan interface can be reached, and `false` otherwise

**wireless→joinNetwork()wireless.joinNetwork()****YWireless**

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

```
def joinNetwork( ssid, securityKey)
```

Remember to call the `saveToFlash( )` method and then to reboot the module to apply this setting.

**Parameters :**

**ssid** the name of the network to connect to  
**securityKey** the network key, as a character string

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

**wireless→load()|wireless.load()****YWireless**

Preloads the wireless lan interface cache with a specified validity duration.

```
def load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI\_SUCCESS when the call succeeds. On failure, throws an exception or returns a negative error code.

**wireless→nextWireless()wireless.nextWireless()****YWireless**

Continues the enumeration of wireless lan interfaces started using `yFirstWireless()`.

```
def nextWireless( )
```

**Returns :**

a pointer to a `YWireless` object, corresponding to a wireless lan interface currently online, or a `null` pointer if there are no more wireless lan interfaces to enumerate.

**wireless→registerValueCallback()  
wireless.registerValueCallback()****YWireless**

Registers the callback function that is invoked on every change of advertised value.

```
def registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

**wireless→set\_logicalName()**  
**wireless→setLogicalName()**  
**wireless.set\_logicalName()**

**YWireless**

Changes the logical name of the wireless lan interface.

```
def set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

`newval` a string corresponding to the logical name of the wireless lan interface.

**Returns :**

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

**wireless→set(userData)**

**YWireless**

**wireless→setUserData(wireless.set(userData))**

---

Stores a user context provided as argument in the userData attribute of the function.

```
def set(userData):
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

# Index

## A

Accelerometer 32  
adhocNetwork, YWireless 1563  
AnButton 74

## B

Blueprint 10

## C

calibrate, YLightSensor 695  
calibrateFromPoints, YAccelerometer 36  
calibrateFromPoints, YCarbonDioxide 116  
calibrateFromPoints, YCompass 184  
calibrateFromPoints, YCurrent 224  
calibrateFromPoints, YGenericSensor 507  
calibrateFromPoints, YGyro 553  
calibrateFromPoints, YHumidity 629  
calibrateFromPoints, YLightSensor 696  
calibrateFromPoints, YMagnetometer 735  
calibrateFromPoints, YPower 906  
calibrateFromPoints, YPressure 949  
calibrateFromPoints, YQt 1049  
calibrateFromPoints,YSensor 1187  
calibrateFromPoints, YTemperature 1261  
calibrateFromPoints, YTilt 1302  
calibrateFromPoints, YVoc 1341  
calibrateFromPoints, YVoltage 1380  
callbackLogin, YNetwork 827  
cancel3DCalibration, YRefFrame 1115  
CarbonDioxide 112  
CheckLogicalName, YAPI 12  
clear, YDisplayLayer 419  
clearConsole, YDisplayLayer 420  
Clock 1084  
ColorLed 151  
Compass 180  
Configuration 1111  
consoleOut, YDisplayLayer 421  
copyLayerContent, YDisplay 375  
Current 220

## D

Data 290, 300, 312  
DataLogger 259  
delayedPulse, YDigitalIO 331  
delayedPulse, YRelay 1151  
delayedPulse, YWatchdog 1519  
describe, YAccelerometer 37  
describe, YAnButton 78  
describe, YCarbonDioxide 117  
describe, YColorLed 154  
describe, YCompass 185

describe, YCurrent 225  
describe, YDataLogger 262  
describe, YDigitalIO 332  
describe, YDisplay 376  
describe, YDualPower 453  
describe, YFiles 478  
describe, YGenericSensor 508  
describe, YGyro 554  
describe, YHubPort 603  
describe, YHumidity 630  
describe, YLed 667  
describe, YLightSensor 697  
describe, YMagnetometer 736  
describe, YModule 783  
describe, YNetwork 828  
describe, YOsControl 882  
describe, YPower 907  
describe, YPressure 950  
describe, YPwmOutput 988  
describe, YPwmPowerSource 1025  
describe, YQt 1050  
describe, YRealTimeClock 1087  
describe, YRefFrame 1116  
describe, YRelay 1152  
describe, YSensor 1188  
describe, YServo 1226  
describe, YTemperature 1262  
describe, YTilt 1303  
describe, YVoc 1342  
describe, YVoltage 1381  
describe, YWakeUpMonitor 1447  
describe, YWakeUpSchedule 1482  
describe, YWatchdog 1520  
describe, YWireless 1564  
Digital 327  
DisableExceptions, YAPI 13  
Display 371  
DisplayLayer 418  
download, YFiles 479  
download, YModule 784  
drawBar, YDisplayLayer 422  
drawBitmap, YDisplayLayer 423  
drawCircle, YDisplayLayer 424  
drawDisc, YDisplayLayer 425  
drawImage, YDisplayLayer 426  
drawPixel, YDisplayLayer 427  
drawRect, YDisplayLayer 428  
drawText, YDisplayLayer 429  
dutyCycleMove, YPwmOutput 989  
Dynamic 3

## E

EnableExceptions, YAPI 14  
Error 7

External 450

## F

fade, YDisplay 377  
Files 3, 475  
FindAccelerometer, YAccelerometer 34  
FindAnButton, YAnButton 76  
FindCarbonDioxide, YCarbonDioxide 114  
FindColorLed, YColorLed 152  
FindCompass, YCompass 182  
FindCurrent, YCurrent 222  
FindDataLogger, YDataLogger 260  
FindDigitalIO, YDigitalIO 329  
FindDisplay, YDisplay 373  
FindDualPower, YDualPower 451  
FindFiles, YFiles 476  
FindGenericSensor, YGenericSensor 505  
FindGyro, YGyro 551  
FindHubPort, YHubPort 601  
FindHumidity, YHumidity 627  
FindLed, YLed 665  
FindLightSensor, YLightSensor 693  
FindMagnetometer, YMagnetometer 733  
FindModule, YModule 781  
FindNetwork, YNetwork 825  
FindOsControl, YOsControl 880  
FindPower, YPower 904  
FindPressure, YPressure 947  
FindPwmOutput, YPwmOutput 986  
FindPwmPowerSource, YPwmPowerSource 1023  
FindQt, YQt 1047  
FindRealTimeClock, YRealTimeClock 1085  
FindRefFrame, YRefFrame 1113  
FindRelay, YRelay 1149  
FindSensor, YSensor 1185  
FindServo,YServo 1224  
FindTemperature, YTemperature 1259  
FindTilt, YTilt 1300  
FindVoc, YVoc 1339  
FindVoltage, YVoltage 1378  
FindVSource, YVSource 1416  
FindWakeUpMonitor, YWakeUpMonitor 1445  
FindWakeUpSchedule, YWakeUpSchedule 1480  
FindWatchdog, YWatchdog 1517  
FindWireless, YWireless 1561  
FirstAccelerometer, YAccelerometer 35  
FirstAnButton, YAnButton 77  
FirstCarbonDioxide, YCarbonDioxide 115  
FirstColorLed, YColorLed 153  
FirstCompass, YCompass 183  
FirstCurrent, YCurrent 223  
FirstDataLogger, YDataLogger 261  
FirstDigitalIO, YDigitalIO 330  
FirstDisplay, YDisplay 374  
FirstDualPower, YDualPower 452  
FirstFiles, YFiles 477  
FirstGenericSensor, YGenericSensor 506  
FirstGyro, YGyro 552

FirstHubPort, YHubPort 602  
FirstHumidity, YHumidity 628  
FirstLed, YLed 666  
FirstLightSensor, YLightSensor 694  
FirstMagnetometer, YMagnetometer 734  
FirstModule, YModule 782  
FirstNetwork, YNetwork 826  
FirstOsControl, YOsControl 881  
FirstPower, YPower 905  
FirstPressure, YPressure 948  
FirstPwmOutput, YPwmOutput 987  
FirstPwmPowerSource, YPwmPowerSource 1024  
FirstQt, YQt 1048  
FirstRealTimeClock, YRealTimeClock 1086  
FirstRefFrame, YRefFrame 1114  
FirstRelay, YRelay 1150  
FirstSensor, YSensor 1186  
FirstServo, YServo 1225  
FirstTemperature, YTemperature 1260  
FirstTilt, YTilt 1301  
FirstVoc, YVoc 1340  
FirstVoltage, YVoltage 1379  
FirstVSource, YVSource 1417  
FirstWakeUpMonitor, YWakeUpMonitor 1446  
FirstWakeUpSchedule, YWakeUpSchedule 1481  
FirstWatchdog, YWatchdog 1518  
FirstWireless, YWireless 1562  
forgetAllDataStreams, YDataLogger 263  
format\_fs, YFiles 480  
Formatted 290  
Frame 1111  
FreeAPI, YAPI 15  
functionCount, YModule 785  
functionId, YModule 786  
functionName, YModule 787  
Functions 11  
functionValue, YModule 788

## G

General 11  
GenericSensor 503  
get\_3DCalibrationHint, YRefFrame 1117  
get\_3DCalibrationLogMsg, YRefFrame 1118  
get\_3DCalibrationProgress, YRefFrame 1119  
get\_3DCalibrationStage, YRefFrame 1120  
get\_3DCalibrationStageProgress, YRefFrame 1121  
get\_adminPassword, YNetwork 829  
get\_advertisedValue, YAccelerometer 38  
get\_advertisedValue, YAnButton 79  
get\_advertisedValue, YCarbonDioxide 118  
get\_advertisedValue, YColorLed 155  
get\_advertisedValue, YCompass 186  
get\_advertisedValue, YCurrent 226  
get\_advertisedValue, YDataLogger 264  
get\_advertisedValue, YDigitalIO 333  
get\_advertisedValue, YDisplay 378  
get\_advertisedValue, YDualPower 454

get\_advertisedValue, YFiles 481  
get\_advertisedValue, YGenericSensor 509  
get\_advertisedValue, YGyro 555  
get\_advertisedValue, YHubPort 604  
get\_advertisedValue, YHumidity 631  
get\_advertisedValue, YLed 668  
get\_advertisedValue, YLightSensor 698  
get\_advertisedValue, YMagnetometer 737  
get\_advertisedValue, YNetwork 830  
get\_advertisedValue, YOsControl 883  
get\_advertisedValue, YPower 908  
get\_advertisedValue, YPressure 951  
get\_advertisedValue, YPwmOutput 990  
get\_advertisedValue, YPwmPowerSource 1026  
get\_advertisedValue, YQt 1051  
get\_advertisedValue, YRealTimeClock 1088  
get\_advertisedValue, YRefFrame 1122  
get\_advertisedValue, YRelay 1153  
get\_advertisedValue, YSensor 1189  
get\_advertisedValue, YServo 1227  
get\_advertisedValue, YTemperature 1263  
get\_advertisedValue, YTilt 1304  
get\_advertisedValue, YVoc 1343  
get\_advertisedValue, YVoltage 1382  
get\_advertisedValue, YVSource 1418  
get\_advertisedValue, YWakeUpMonitor 1448  
get\_advertisedValue, YWakeUpSchedule 1483  
get\_advertisedValue, YWatchdog 1521  
get\_advertisedValue, YWireless 1565  
get\_analogCalibration, YAnButton 80  
get\_autoStart, YDataLogger 265  
get\_autoStart, YWatchdog 1522  
get\_averageValue, YDataRun 290  
get\_averageValue, YDataStream 313  
get\_averageValue, YMeasure 773  
get\_baudRate, YHubPort 605  
get\_beacon, YModule 789  
get\_bearing, YRefFrame 1123  
get\_bitDirection, YDigitalIO 334  
get\_bitOpenDrain, YDigitalIO 335  
get\_bitPolarity, YDigitalIO 336  
get\_bitState, YDigitalIO 337  
get\_blinking, YLed 669  
get\_brightness, YDisplay 379  
get\_calibratedValue, YAnButton 81  
get\_calibrationMax, YAnButton 82  
get\_calibrationMin, YAnButton 83  
get\_callbackCredentials, YNetwork 831  
get\_callbackEncoding, YNetwork 832  
get\_callbackMaxDelay, YNetwork 833  
get\_callbackMethod, YNetwork 834  
get\_callbackMinDelay, YNetwork 835  
get\_callbackUrl, YNetwork 836  
get\_channel, YWireless 1566  
get\_columnCount, YDataStream 314  
get\_columnNames, YDataStream 315  
get\_cosPhi, YPower 909  
get\_countdown, YRelay 1154  
get\_countdown, YWatchdog 1523  
get\_currentRawValue, YAccelerometer 39  
get\_currentRawValue, YCarbonDioxide 119  
get\_currentRawValue, YCompass 187  
get\_currentRawValue, YCurrent 227  
get\_currentRawValue, YGenericSensor 510  
get\_currentRawValue, YGyro 556  
get\_currentRawValue, YHumidity 632  
get\_currentRawValue, YLightSensor 699  
get\_currentRawValue, YMagnetometer 738  
get\_currentRawValue, YPower 910  
get\_currentRawValue, YPressure 952  
get\_currentRawValue, YQt 1052  
get\_currentRawValue, YSensor 1190  
get\_currentRawValue, YTemperature 1264  
get\_currentRawValue, YTilt 1305  
get\_currentRawValue, YVoc 1344  
get\_currentRawValue, YVoltage 1383  
get\_currentRunIndex, YDataLogger 266  
get\_currentValue, YAccelerometer 40  
get\_currentValue, YCarbonDioxide 120  
get\_currentValue, YCompass 188  
get\_currentValue, YCurrent 228  
get\_currentValue, YGenericSensor 511  
get\_currentValue, YGyro 557  
get\_currentValue, YHumidity 633  
get\_currentValue, YLightSensor 700  
get\_currentValue, YMagnetometer 739  
get\_currentValue, YPower 911  
get\_currentValue, YPressure 953  
get\_currentValue, YQt 1053  
get\_currentValue, YSensor 1191  
get\_currentValue, YTemperature 1265  
get\_currentValue, YTilt 1306  
get\_currentValue, YVoc 1345  
get\_currentValue, YVoltage 1384  
get\_data, YDataStream 316  
get\_dataRows, YDataStream 317  
get\_dataSamplesIntervalMs, YDataStream 318  
get\_dataSets, YDataLogger 267  
get\_dataStreams, YDataLogger 268  
get\_dateTime, YRealTimeClock 1089  
get\_detectedWlans, YWireless 1567  
get\_discoverable, YNetwork 837  
get\_display, YDisplayLayer 430  
get\_displayHeight, YDisplay 380  
get\_displayHeight, YDisplayLayer 431  
get\_displayLayer, YDisplay 381  
get\_displayType, YDisplay 382  
get\_displayWidth, YDisplay 383  
get\_displayWidth, YDisplayLayer 432  
get\_duration, YDataRun 291  
get\_duration, YDataStream 319  
get\_dutyCycle, YPwmOutput 991  
get\_dutyCycleAtPowerOn, YPwmOutput 992  
get\_enabled, YDisplay 384  
get\_enabled, YHubPort 606  
get\_enabled, YPwmOutput 993  
get\_enabled, YServo 1228  
get\_enabledAtPowerOn, YPwmOutput 994

get\_enabledAtPowerOn, YServo 1229  
get\_endTimeUTC, YDataSet 301  
get\_endTimeUTC, YMeasure 774  
get\_errorMessage, YAccelerometer 41  
get\_errorMessage, YAnButton 84  
get\_errorMessage, YCarbonDioxide 121  
get\_errorMessage, YColorLed 156  
get\_errorMessage, YCompass 189  
get\_errorMessage, YCurrent 229  
get\_errorMessage, YDataLogger 269  
get\_errorMessage, YDigitalIO 338  
get\_errorMessage, YDisplay 385  
get\_errorMessage, YDualPower 455  
get\_errorMessage, YFiles 482  
get\_errorMessage, YGenericSensor 512  
get\_errorMessage, YGyro 558  
get\_errorMessage, YHubPort 607  
get\_errorMessage, YHumidity 634  
get\_errorMessage, YLed 670  
get\_errorMessage, YLightSensor 701  
get\_errorMessage, YMagnetometer 740  
get\_errorMessage, YModule 790  
get\_errorMessage, YNetwork 838  
get\_errorMessage, YOsControl 884  
get\_errorMessage, YPower 912  
get\_errorMessage, YPressure 954  
get\_errorMessage, YPwmOutput 995  
get\_errorMessage, YPwmPowerSource 1027  
get\_errorMessage, YQt 1054  
get\_errorMessage, YRealTimeClock 1090  
get\_errorMessage, YRefFrame 1124  
get\_errorMessage, YRelay 1155  
get\_errorMessage, YSensor 1192  
get\_errorMessage,YServo 1230  
get\_errorMessage, YTemperature 1266  
get\_errorMessage, YTilt 1307  
get\_errorMessage, YVoc 1346  
get\_errorMessage, YVoltage 1385  
get\_errorMessage, YVSource 1419  
get\_errorMessage, YWakeUpMonitor 1449  
get\_errorMessage, YWakeUpSchedule 1484  
get\_errorMessage, YWatchdog 1524  
get\_errorMessage, YWireless 1568  
get\_errorType, YAccelerometer 42  
get\_errorType, YAnButton 85  
get\_errorType, YCarbonDioxide 122  
get\_errorType, YColorLed 157  
get\_errorType, YCompass 190  
get\_errorType, YCurrent 230  
get\_errorType, YDataLogger 270  
get\_errorType, YDigitalIO 339  
get\_errorType, YDisplay 386  
get\_errorType, YDualPower 456  
get\_errorType, YFiles 483  
get\_errorType, YGenericSensor 513  
get\_errorType, YGyro 559  
get\_errorType, YHubPort 608  
get\_errorType, YHumidity 635  
get\_errorType, YLed 671  
get\_errorType, YLightSensor 702  
get\_errorType, YMagnetometer 741  
get\_errorType, YModule 791  
get\_errorType, YNetwork 839  
get\_errorType, YOsControl 885  
get\_errorType, YPower 913  
get\_errorType, YPressure 955  
get\_errorType, YPwmOutput 996  
get\_errorType, YPwmPowerSource 1028  
get\_errorType, YQt 1055  
get\_errorType, YRealTimeClock 1091  
get\_errorType, YRefFrame 1125  
get\_errorType, YRelay 1156  
get\_errorType, YSensor 1193  
get\_errorType, YServo 1231  
get\_errorType, YTemperature 1267  
get\_errorType, YTilt 1308  
get\_errorType, YVoc 1347  
get\_errorType, YVoltage 1386  
get\_errorType, YVSource 1420  
get\_errorType, YWakeUpMonitor 1450  
get\_errorType, YWakeUpSchedule 1485  
get\_errorType, YWatchdog 1525  
get\_errorType, YWireless 1569  
get\_extPowerFailure, YVSource 1421  
get\_extVoltage, YDualPower 457  
get\_failure, YVSource 1422  
get\_filesCount, YFiles 484  
get\_firmwareRelease, YModule 792  
get\_freeSpace, YFiles 485  
get\_frequency, YPwmOutput 997  
get\_friendlyName, YAccelerometer 43  
get\_friendlyName, YAnButton 86  
get\_friendlyName, YCarbonDioxide 123  
get\_friendlyName, YColorLed 158  
get\_friendlyName, YCompass 191  
get\_friendlyName, YCurrent 231  
get\_friendlyName, YDataLogger 271  
get\_friendlyName, YDigitalIO 340  
get\_friendlyName, YDisplay 387  
get\_friendlyName, YDualPower 458  
get\_friendlyName, YFiles 486  
get\_friendlyName, YGenericSensor 514  
get\_friendlyName, YGyro 560  
get\_friendlyName, YHubPort 609  
get\_friendlyName, YHumidity 636  
get\_friendlyName, YLed 672  
get\_friendlyName, YLightSensor 703  
get\_friendlyName, YMagnetometer 742  
get\_friendlyName, YNetwork 840  
get\_friendlyName, YOsControl 886  
get\_friendlyName, YPower 914  
get\_friendlyName, YPressure 956  
get\_friendlyName, YPwmOutput 998  
get\_friendlyName, YPwmPowerSource 1029  
get\_friendlyName, YQt 1056  
get\_friendlyName, YRealTimeClock 1092  
get\_friendlyName, YRefFrame 1126  
get\_friendlyName, YRelay 1157

get\_friendlyName, YSensor 1194  
get\_friendlyName,YServo 1232  
get\_friendlyName,YTemperature 1268  
get\_friendlyName,YTilt 1309  
get\_friendlyName,YVoc 1348  
get\_friendlyName,YVoltage 1387  
get\_friendlyName,YWakeUpMonitor 1451  
get\_friendlyName,YWakeUpSchedule 1486  
get\_friendlyName,YWatchdog 1526  
get\_friendlyName,YWireless 1570  
get\_functionDescriptor, YAccelerometer 44  
get\_functionDescriptor, YAnButton 87  
get\_functionDescriptor, YCarbonDioxide 124  
get\_functionDescriptor, YColorLed 159  
get\_functionDescriptor, YCompass 192  
get\_functionDescriptor, YCurrent 232  
get\_functionDescriptor, YDataLogger 272  
get\_functionDescriptor, YDigitalIO 341  
get\_functionDescriptor, YDisplay 388  
get\_functionDescriptor, YDualPower 459  
get\_functionDescriptor, YFiles 487  
get\_functionDescriptor, YGenericSensor 515  
get\_functionDescriptor, YGyro 561  
get\_functionDescriptor, YHubPort 610  
get\_functionDescriptor, YHumidity 637  
get\_functionDescriptor, YLed 673  
get\_functionDescriptor, YLightSensor 704  
get\_functionDescriptor, YMagnetometer 743  
get\_functionDescriptor, YNetwork 841  
get\_functionDescriptor, YOsControl 887  
get\_functionDescriptor, YPower 915  
get\_functionDescriptor, YPressure 957  
get\_functionDescriptor, YPwmOutput 999  
get\_functionDescriptor, YPwmPowerSource 1030  
get\_functionDescriptor, YQt 1057  
get\_functionDescriptor, YRealTimeClock 1093  
get\_functionDescriptor, YRefFrame 1127  
get\_functionDescriptor, YRelay 1158  
get\_functionDescriptor,YSensor 1195  
get\_functionDescriptor,YServo 1233  
get\_functionDescriptor,YTemperature 1269  
get\_functionDescriptor,YTilt 1310  
get\_functionDescriptor,YVoc 1349  
get\_functionDescriptor,YVoltage 1388  
get\_functionDescriptor,YVSource 1423  
get\_functionDescriptor,YWakeUpMonitor 1452  
get\_functionDescriptor,YWakeUpSchedule 1487  
get\_functionDescriptor,YWatchdog 1527  
get\_functionDescriptor,YWireless 1571  
get\_functionId, YAccelerometer 45  
get\_functionId, YAnButton 88  
get\_functionId, YCarbonDioxide 125  
get\_functionId, YColorLed 160  
get\_functionId, YCompass 193  
get\_functionId, YCurrent 233  
get\_functionId, YDataLogger 273  
get\_functionId, YDataSet 302  
get\_functionId, YDigitalIO 342  
get\_functionId, YDisplay 389  
get\_functionId, YDualPower 460  
get\_functionId, YFiles 488  
get\_functionId, YGenericSensor 516  
get\_functionId, YGyro 562  
get\_functionId, YHubPort 611  
get\_functionId, YHumidity 638  
get\_functionId, YLed 674  
get\_functionId, YLightSensor 705  
get\_functionId, YMagnetometer 744  
get\_functionId, YNetwork 842  
get\_functionId, YOsControl 888  
get\_functionId, YPower 916  
get\_functionId, YPressure 958  
get\_functionId, YPwmOutput 1000  
get\_functionId, YPwmPowerSource 1031  
get\_functionId, YQt 1058  
get\_functionId, YRealTimeClock 1094  
get\_functionId, YRefFrame 1128  
get\_functionId, YRelay 1159  
get\_functionId,YSensor 1196  
get\_functionId,YServo 1234  
get\_functionId,YTemperature 1270  
get\_functionId,YTilt 1311  
get\_functionId,YVoc 1350  
get\_functionId,YVoltage 1389  
get\_functionId,YWakeUpMonitor 1453  
get\_functionId,YWakeUpSchedule 1488  
get\_functionId,YWatchdog 1528  
get\_functionId,YWireless 1572  
get\_hardwareId, YAccelerometer 46  
get\_hardwareId, YAnButton 89  
get\_hardwareId, YCarbonDioxide 126  
get\_hardwareId, YColorLed 161  
get\_hardwareId, YCompass 194  
get\_hardwareId, YCurrent 234  
get\_hardwareId, YDataLogger 274  
get\_hardwareId, YDataSet 303  
get\_hardwareId, YDigitalIO 343  
get\_hardwareId, YDisplay 390  
get\_hardwareId, YDualPower 461  
get\_hardwareId, YFiles 489  
get\_hardwareId, YGenericSensor 517  
get\_hardwareId, YGyro 563  
get\_hardwareId, YHubPort 612  
get\_hardwareId, YHumidity 639  
get\_hardwareId, YLed 675  
get\_hardwareId, YLightSensor 706  
get\_hardwareId, YMagnetometer 745  
get\_hardwareId, YModule 793  
get\_hardwareId, YNetwork 843  
get\_hardwareId, YOsControl 889  
get\_hardwareId, YPower 917  
get\_hardwareId, YPressure 959  
get\_hardwareId, YPwmOutput 1001  
get\_hardwareId, YPwmPowerSource 1032  
get\_hardwareId, YQt 1059  
get\_hardwareId, YRealTimeClock 1095  
get\_hardwareId, YRefFrame 1129  
get\_hardwareId, YRelay 1160

get\_hardwareId, YSensor 1197  
get\_hardwareId, YServo 1235  
get\_hardwareId, YTemperature 1271  
get\_hardwareId, YTilt 1312  
get\_hardwareId, YVoc 1351  
get\_hardwareId, YVoltage 1390  
get\_hardwareId, YWakeUpMonitor 1454  
get\_hardwareId, YWakeUpSchedule 1489  
get\_hardwareId, YWatchdog 1529  
get\_hardwareId, YWireless 1573  
get\_heading, YGyro 564  
get\_highestValue, YAccelerometer 47  
get\_highestValue, YCarbonDioxide 127  
get\_highestValue, YCompass 195  
get\_highestValue, YCurrent 235  
get\_highestValue, YGenericSensor 518  
get\_highestValue, YGyro 565  
get\_highestValue, YHumidity 640  
get\_highestValue, YLightSensor 707  
get\_highestValue, YMagnetometer 746  
get\_highestValue, YPower 918  
get\_highestValue, YPressure 960  
get\_highestValue, YQt 1060  
get\_highestValue, YSensor 1198  
get\_highestValue, YTemperature 1272  
get\_highestValue, YTilt 1313  
get\_highestValue, YVoc 1352  
get\_highestValue, YVoltage 1391  
get\_hours, YWakeUpSchedule 1490  
get\_hslColor, YColorLed 162  
get\_icon2d, YModule 794  
get\_ipAddress, YNetwork 844  
get\_isPressed, YAnButton 90  
get\_lastLogs, YModule 795  
get\_lastTimePressed, YAnButton 91  
get\_lastTimeReleased, YAnButton 92  
get\_layerCount, YDisplay 391  
get\_layerHeight, YDisplay 392  
get\_layerHeight, YDisplayLayer 433  
get\_layerWidth, YDisplay 393  
get\_layerWidth, YDisplayLayer 434  
get\_linkQuality, YWireless 1574  
get\_list, YFiles 490  
get\_logFrequency, YAccelerometer 48  
get\_logFrequency, YCarbonDioxide 128  
get\_logFrequency, YCompass 196  
get\_logFrequency, YCurrent 236  
get\_logFrequency, YGenericSensor 519  
get\_logFrequency, YGyro 566  
get\_logFrequency, YHumidity 641  
get\_logFrequency, YLightSensor 708  
get\_logFrequency, YMagnetometer 747  
get\_logFrequency, YPower 919  
get\_logFrequency, YPressure 961  
get\_logFrequency, YQt 1061  
get\_logFrequency, YSensor 1199  
get\_logFrequency, YTemperature 1273  
get\_logFrequency, YTilt 1314  
get\_logFrequency, YVoc 1353  
get\_logFrequency, YVoltage 1392  
get\_logicalName, YAccelerometer 49  
get\_logicalName, YAnButton 93  
get\_logicalName, YCarbonDioxide 129  
get\_logicalName, YColorLed 163  
get\_logicalName, YCompass 197  
get\_logicalName, YCurrent 237  
get\_logicalName, YDataLogger 275  
get\_logicalName, YDigitalIO 344  
get\_logicalName, YDisplay 394  
get\_logicalName, YDualPower 462  
get\_logicalName, YFiles 491  
get\_logicalName, YGenericSensor 520  
get\_logicalName, YGyro 567  
get\_logicalName, YHubPort 613  
get\_logicalName, YHumidity 642  
get\_logicalName, YLed 676  
get\_logicalName, YLightSensor 709  
get\_logicalName, YMagnetometer 748  
get\_logicalName, YModule 796  
get\_logicalName, YNetwork 845  
get\_logicalName, YOsControl 890  
get\_logicalName, YPower 920  
get\_logicalName, YPressure 962  
get\_logicalName, YPwmOutput 1002  
get\_logicalName, YPwmPowerSource 1033  
get\_logicalName, YQt 1062  
get\_logicalName, YRealTimeClock 1096  
get\_logicalName, YRefFrame 1130  
get\_logicalName, YRelay 1161  
get\_logicalName, YSensor 1200  
get\_logicalName, YServo 1236  
get\_logicalName, YTemperature 1274  
get\_logicalName, YTilt 1315  
get\_logicalName, YVoc 1354  
get\_logicalName, YVoltage 1393  
get\_logicalName, YVSource 1424  
get\_logicalName, YWakeUpMonitor 1455  
get\_logicalName, YWakeUpSchedule 1491  
get\_logicalName, YWatchdog 1530  
get\_logicalName, YWireless 1575  
get\_lowestValue, YAccelerometer 50  
get\_lowestValue, YCarbonDioxide 130  
get\_lowestValue, YCompass 198  
get\_lowestValue, YCurrent 238  
get\_lowestValue, YGenericSensor 521  
get\_lowestValue, YGyro 568  
get\_lowestValue, YHumidity 643  
get\_lowestValue, YLightSensor 710  
get\_lowestValue, YMagnetometer 749  
get\_lowestValue, YPower 921  
get\_lowestValue, YPressure 963  
get\_lowestValue, YQt 1063  
get\_lowestValue, YSensor 1201  
get\_lowestValue, YTemperature 1275  
get\_lowestValue, YTilt 1316  
get\_lowestValue, YVoc 1355  
get\_lowestValue, YVoltage 1394  
get\_luminosity, YLed 677

get\_luminosity, YModule 797  
get\_macAddress, YNetwork 846  
get\_magneticHeading, YCompass 199  
get\_maxTimeOnStateA, YRelay 1162  
get\_maxTimeOnStateA, YWatchdog 1531  
get\_maxTimeOnStateB, YRelay 1163  
get\_maxTimeOnStateB, YWatchdog 1532  
get\_maxValue, YDataRun 292  
get\_maxValue, YDataStream 320  
get\_maxValue, YMeasure 775  
get\_measureNames, YDataRun 293  
get\_measures, YDataSet 304  
get\_message, YWireless 1576  
get\_meter, YPower 922  
get\_meterTimer, YPower 923  
get\_minutes, YWakeUpSchedule 1492  
get\_minutesA, YWakeUpSchedule 1493  
get\_minutesB, YWakeUpSchedule 1494  
get\_minValue, YDataRun 294  
get\_minValue, YDataStream 321  
get\_minValue, YMeasure 776  
get\_module, YAccelerometer 51  
get\_module, YAnButton 94  
get\_module, YCarbonDioxide 131  
get\_module, YColorLed 164  
get\_module, YCompass 200  
get\_module, YCurrent 239  
get\_module, YDataLogger 276  
get\_module, YDigitalIO 345  
get\_module, YDisplay 395  
get\_module, YDualPower 463  
get\_module, YFiles 492  
get\_module, YGenericSensor 522  
get\_module, YGyro 569  
get\_module, YHubPort 614  
get\_module, YHumidity 644  
get\_module, YLed 678  
get\_module, YLightSensor 711  
get\_module, YMagnetometer 750  
get\_module, YNetwork 847  
get\_module, YOsControl 891  
get\_module, YPower 924  
get\_module, YPressure 964  
get\_module, YPwmOutput 1003  
get\_module, YPwmPowerSource 1034  
get\_module, YQt 1064  
get\_module, YRealTimeClock 1097  
get\_module, YRefFrame 1131  
get\_module, YRelay 1164  
get\_module,YSensor 1202  
get\_module, YServo 1237  
get\_module, YTemperature 1276  
get\_module, YTilt 1317  
get\_module, YVoc 1356  
get\_module, YVoltage 1395  
get\_module, YVSource 1425  
get\_module, YWakeUpMonitor 1456  
get\_module, YWakeUpSchedule 1495  
get\_module, YWatchdog 1533  
get\_module, YWireless 1577  
get\_monthDays, YWakeUpSchedule 1496  
get\_months, YWakeUpSchedule 1497  
get\_mountOrientation, YRefFrame 1132  
get\_mountPosition, YRefFrame 1133  
get\_neutral, YServo 1238  
get\_nextOccurrence, YWakeUpSchedule 1498  
get\_nextWakeUp, YWakeUpMonitor 1457  
get\_orientation, YDisplay 396  
get\_output, YRelay 1165  
get\_output, YWatchdog 1534  
get\_outputVoltage, YDigitalIO 346  
get\_overCurrent, YVSource 1426  
get\_overHeat, YVSource 1427  
get\_overLoad, YVSource 1428  
get\_period, YPwmOutput 1004  
get\_persistentSettings, YModule 798  
get\_pitch, YGyro 570  
get\_poeCurrent, YNetwork 848  
get\_portDirection, YDigitalIO 347  
get\_portOpenDrain, YDigitalIO 348  
get\_portPolarity, YDigitalIO 349  
get\_portSize, YDigitalIO 350  
get\_portState, YDigitalIO 351  
get\_portState, YHubPort 615  
get\_position, YServo 1239  
get\_positionAtPowerOn, YServo 1240  
get\_power, YLed 679  
get\_powerControl, YDualPower 464  
get\_powerDuration, YWakeUpMonitor 1458  
get\_powerMode, YPwmPowerSource 1035  
get\_powerState, YDualPower 465  
get\_preview, YDataSet 305  
get\_primaryDNS, YNetwork 849  
get\_productId, YModule 799  
get\_productName, YModule 800  
get\_productRelease, YModule 801  
get\_progress, YDataSet 306  
get\_pulseCounter, YAnButton 95  
get\_pulseDuration, YPwmOutput 1005  
get\_pulseTimer, YAnButton 96  
get\_pulseTimer, YRelay 1166  
get\_pulseTimer, YWatchdog 1535  
get\_quaternionW, YGyro 571  
get\_quaternionX, YGyro 572  
get\_quaternionY, YGyro 573  
get\_quaternionZ, YGyro 574  
get\_range, YServo 1241  
get\_rawValue, YAnButton 97  
get\_readiness, YNetwork 850  
get\_rebootCountdown, YModule 802  
get\_recordedData, YAccelerometer 52  
get\_recordedData, YCarbonDioxide 132  
get\_recordedData, YCompass 201  
get\_recordedData, YCurrent 240  
get\_recordedData, YGenericSensor 523  
get\_recordedData, YGyro 575  
get\_recordedData, YHumidity 645  
get\_recordedData, YLightSensor 712

get\_recordedData, YMagnetometer 751  
get\_recordedData, YPower 925  
get\_recordedData, YPressure 965  
get\_recordedData, YQt 1065  
get\_recordedData, YSensor 1203  
get\_recordedData, YTemperature 1277  
get\_recordedData, YTilt 1318  
get\_recordedData, YVoc 1357  
get\_recordedData, YVoltage 1396  
get\_recording, YDataStream 277  
get\_regulationFailure, YVSource 1429  
get\_reportFrequency, YAccelerometer 53  
get\_reportFrequency, YCarbonDioxide 133  
get\_reportFrequency, YCompass 202  
get\_reportFrequency, YCurrent 241  
get\_reportFrequency, YGenericSensor 524  
get\_reportFrequency, YGyro 576  
get\_reportFrequency, YHumidity 646  
get\_reportFrequency, YLightSensor 713  
get\_reportFrequency, YMagnetometer 752  
get\_reportFrequency, YPower 926  
get\_reportFrequency, YPressure 966  
get\_reportFrequency, YQt 1066  
get\_reportFrequency, YSensor 1204  
get\_reportFrequency, YTemperature 1278  
get\_reportFrequency, YTilt 1319  
get\_reportFrequency, YVoc 1358  
get\_reportFrequency, YVoltage 1397  
get\_resolution, YAccelerometer 54  
get\_resolution, YCarbonDioxide 134  
get\_resolution, YCompass 203  
get\_resolution, YCurrent 242  
get\_resolution, YGenericSensor 525  
get\_resolution, YGyro 577  
get\_resolution, YHumidity 647  
get\_resolution, YLightSensor 714  
get\_resolution, YMagnetometer 753  
get\_resolution, YPower 927  
get\_resolution, YPressure 967  
get\_resolution, YQt 1067  
get\_resolution, YSensor 1205  
get\_resolution, YTemperature 1279  
get\_resolution, YTilt 1320  
get\_resolution, YVoc 1359  
get\_resolution, YVoltage 1398  
get\_rgbColor, YColorLed 165  
get\_rgbColorAtPowerOn, YColorLed 166  
get\_roll, YGyro 578  
get\_router, YNetwork 851  
getRowCount, YDataStream 322  
get\_runIndex, YDataStream 323  
get\_running, YWatchdog 1536  
get\_secondaryDNS, YNetwork 852  
get\_security, YWireless 1578  
get\_sensitivity, YAnButton 98  
get\_sensorType, YTemperature 1280  
get\_serialNumber, YModule 803  
get\_shutdownCountdown, YOsControl 892  
get\_signalRange, YGenericSensor 526  
get\_signalUnit, YGenericSensor 527  
get\_signalValue, YGenericSensor 528  
get\_sleepCountdown, YWakeUpMonitor 1459  
get\_ssid, YWireless 1579  
get\_startTime, YDataStream 324  
get\_startTimeUTC, YDataRun 295  
get\_startTimeUTC, YDataSet 307  
get\_startTimeUTC, YDataStream 325  
get\_startTimeUTC, YMeasure 777  
get\_startupSeq, YDisplay 397  
get\_state, YRelay 1167  
get\_state, YWatchdog 1537  
get\_stateAtPowerOn, YRelay 1168  
get\_stateAtPowerOn, YWatchdog 1538  
get\_subnetMask, YNetwork 853  
get\_summary, YDataSet 308  
get\_timeSet, YRealTimeClock 1098  
get\_timeUTC, YDataStream 278  
get\_triggerDelay, YWatchdog 1539  
get\_triggerDuration, YWatchdog 1540  
get\_unit, YAccelerometer 55  
get\_unit, YCarbonDioxide 135  
get\_unit, YCompass 204  
get\_unit, YCurrent 243  
get\_unit, YDataSet 309  
get\_unit, YGenericSensor 529  
get\_unit, YGyro 579  
get\_unit, YHumidity 648  
get\_unit, YLightSensor 715  
get\_unit, YMagnetometer 754  
get\_unit, YPower 928  
get\_unit, YPressure 968  
get\_unit, YQt 1068  
get\_unit, YSensor 1206  
get\_unit, YTemperature 1281  
get\_unit, YTilt 1321  
get\_unit, YVoc 1360  
get\_unit, YVoltage 1399  
get\_unit, YVSource 1430  
get\_unixTime, YRealTimeClock 1099  
get\_upTime, YModule 804  
get\_usbBandwidth, YModule 805  
get\_usbCurrent, YModule 806  
get\_userData, YAccelerometer 56  
get\_userData, YAnButton 99  
get\_userData, YCarbonDioxide 136  
get\_userData, YColorLed 167  
get\_userData, YCompass 205  
get\_userData, YCurrent 244  
get\_userData, YDataStream 279  
get\_userData, YDigitalIO 352  
get\_userData, YDisplay 398  
get\_userData, YDualPower 466  
get\_userData, YFiles 493  
get\_userData, YGenericSensor 530  
get\_userData, YGyro 580  
get\_userData, YHubPort 616  
get\_userData, YHumidity 649  
get\_userData, YLed 680

get(userData, YLightSensor 716  
get(userData, YMagnetometer 755  
get(userData, YModule 807  
get(userData, YNetwork 854  
get(userData, YOsControl 893  
get(userData, YPower 929  
get(userData, YPressure 969  
get(userData, YPwmOutput 1006  
get(userData, YPwmPowerSource 1036  
get(userData, YQt 1069  
get(userData, YRealTimeClock 1100  
get(userData, YRefFrame 1134  
get(userData, YRelay 1169  
get(userData, YSensor 1207  
get(userData, YServo 1242  
get(userData, YTTemperature 1282  
get(userData, YTilt 1322  
get(userData, YVoc 1361  
get(userData, YVoltage 1400  
get(userData, YVSource 1431  
get(userData, YWakeUpMonitor 1460  
get(userData, YWakeUpSchedule 1499  
get(userData, YWatchdog 1541  
get(userData, YWireless 1580  
get(userPassword, YNetwork 855  
get(utcOffset, YRealTimeClock 1101  
get(valueCount, YDataRun 296  
get(valueInterval, YDataRun 297  
get(valueRange, YGenericSensor 531  
get(voltage, YVSource 1432  
get(wakeUpReason, YWakeUpMonitor 1461  
get(wakeUpState, YWakeUpMonitor 1462  
get(weekDays, YWakeUpSchedule 1500  
get(wwwWatchdogDelay, YNetwork 856  
get(xValue, YAccelerometer 57  
get(xValue, YGyro 581  
get(xValue, YMagnetometer 756  
get(yValue, YAccelerometer 58  
get(yValue, YGyro 582  
get(yValue, YMagnetometer 757  
get(zValue, YAccelerometer 59  
get(zValue, YGyro 583  
get(zValue, YMagnetometer 758  
GetAPIVersion, YAPI 16  
GetTickCount, YAPI 17  
Gyroscope 549

## H

HandleEvents, YAPI 18  
hide, YDisplayLayer 435  
hslMove, YColorLed 168  
Humidity 625

## I

InitAPI, YAPI 19  
Interface 32, 74, 112, 151, 180, 220, 259, 327,  
371, 418, 450, 475, 503, 549, 600, 625, 664,  
691, 731, 779, 822, 902, 945, 984, 1022, 1045,

1084, 1147, 1183, 1222, 1257, 1298, 1337,  
1376, 1415, 1443, 1478, 1515, 1560  
Introduction 1  
isOnline, YAccelerometer 60  
isOnline, YAnButton 100  
isOnline, YCarbonDioxide 137  
isOnline, YColorLed 169  
isOnline, YCompass 206  
isOnline, YCurrent 245  
isOnline, YDataLogger 280  
isOnline, YDigitalIO 353  
isOnline, YDisplay 399  
isOnline, YDualPower 467  
isOnline, YFiles 494  
isOnline, YGenericSensor 532  
isOnline, YGyro 584  
isOnline, YHubPort 617  
isOnline, YHumidity 650  
isOnline, YLed 681  
isOnline, YLightSensor 717  
isOnline, YMagnetometer 759  
isOnline, YModule 808  
isOnline, YNetwork 857  
isOnline, YOsControl 894  
isOnline, YPower 930  
isOnline, YPressure 970  
isOnline, YPwmOutput 1007  
isOnline, YPwmPowerSource 1037  
isOnline, YQt 1070  
isOnline, YRealTimeClock 1102  
isOnline, YRefFrame 1135  
isOnline, YRelay 1170  
isOnline, YSensor 1208  
isOnline, YServo 1243  
isOnline, YTTemperature 1283  
isOnline, YTilt 1323  
isOnline, YVoc 1362  
isOnline, YVoltage 1401  
isOnline, YVSource 1433  
isOnline, YWakeUpMonitor 1463  
isOnline, YWakeUpSchedule 1501  
isOnline, YWatchdog 1542  
isOnline, YWireless 1581

## J

joinNetwork, YWireless 1582

## L

Library 3  
LightSensor 691  
lineTo, YDisplayLayer 436  
load, YAccelerometer 61  
load, YAnButton 101  
load, YCarbonDioxide 138  
load, YColorLed 170  
load, YCompass 207  
load, YCurrent 246  
load, YDataLogger 281

load, YDigitalIO 354  
load, YDisplay 400  
load, YDualPower 468  
load, YFiles 495  
load, YGenericSensor 533  
load, YGyro 585  
load, YHubPort 618  
load, YHumidity 651  
load, YLed 682  
load, YLightSensor 718  
load, YMagnetometer 760  
load, YModule 809  
load, YNetwork 858  
load, YOsControl 895  
load, YPower 931  
load, YPressure 971  
load, YPwmOutput 1008  
load, YPwmPowerSource 1038  
load, YQt 1071  
load, YRealTimeClock 1103  
load, YRefFrame 1136  
load, YRelay 1171  
load, YSensor 1209  
load, YServo 1244  
load, YTemperature 1284  
load, YTilt 1324  
load, YVoc 1363  
load, YVoltage 1402  
load, YVSource 1434  
load, YWakeUpMonitor 1464  
load, YWakeUpSchedule 1502  
load, YWatchdog 1543  
load, YWireless 1583  
loadCalibrationPoints, YAccelerometer 62  
loadCalibrationPoints, YCarbonDioxide 139  
loadCalibrationPoints, YCompass 208  
loadCalibrationPoints, YCurrent 247  
loadCalibrationPoints, YGenericSensor 534  
loadCalibrationPoints, YGyro 586  
loadCalibrationPoints, YHumidity 652  
loadCalibrationPoints, YLightSensor 719  
loadCalibrationPoints, YMagnetometer 761  
loadCalibrationPoints, YPower 932  
loadCalibrationPoints, YPressure 972  
loadCalibrationPoints, YQt 1072  
loadCalibrationPoints, YSensor 1210  
loadCalibrationPoints, YTemperature 1285  
loadCalibrationPoints, YTilt 1325  
loadCalibrationPoints, YVoc 1364  
loadCalibrationPoints, YVoltage 1403  
loadMore, YDataSet 310

## M

Magnetometer 731  
Measured 773  
Module 5, 779  
more3DCalibration, YRefFrame 1137  
move, YServo 1245  
moveTo, YDisplayLayer 437

## N

Network 822  
newSequence, YDisplay 401  
nextAccelerometer, YAccelerometer 63  
nextAnButton, YAnButton 102  
nextCarbonDioxide, YCarbonDioxide 140  
nextColorLed, YColorLed 171  
nextCompass, YCompass 209  
nextCurrent, YCurrent 248  
nextDataLogger, YDataLogger 282  
nextDigitalIO, YDigitalIO 355  
nextDisplay, YDisplay 402  
nextDualPower, YDualPower 469  
nextFiles, YFiles 496  
nextGenericSensor, YGenericSensor 535  
nextGyro, YGyro 587  
nextHubPort, YHubPort 619  
nextHumidity, YHumidity 653  
nextLed, YLed 683  
nextLightSensor, YLightSensor 720  
nextMagnetometer, YMagnetometer 762  
nextModule, YModule 810  
nextNetwork, YNetwork 859  
nextOsControl, YOsControl 896  
nextPower, YPower 933  
nextPressure, YPressure 973  
nextPwmOutput, YPwmOutput 1009  
nextPwmPowerSource, YPwmPowerSource 1039  
nextQt, YQt 1073  
nextRealTimeClock, YRealTimeClock 1104  
nextRefFrame, YRefFrame 1138  
nextRelay, YRelay 1172  
nextSensor, YSensor 1211  
nextServo, YServo 1246  
nextTemperature, YTemperature 1286  
nextTilt, YTilt 1326  
nextVoc, YVoc 1365  
nextVoltage, YVoltage 1404  
nextVSource, YVSource 1435  
nextWakeUpMonitor, YWakeUpMonitor 1465  
nextWakeUpSchedule, YWakeUpSchedule 1503  
nextWatchdog, YWatchdog 1544  
nextWireless, YWireless 1584

## O

Object 418

## P

pauseSequence, YDisplay 403  
ping, YNetwork 860  
playSequence, YDisplay 404  
Port 600  
Power 450, 902  
PreregisterHub, YAPI 20  
Pressure 945

pulse, YDigitalIO 356  
pulse, YRelay 1173  
pulse, YVSource 1436  
pulse, YWatchdog 1545  
pulseDurationMove, YPwmOutput 1010  
PwmPowerSource 1022  
Python 3

## Q

Quaternion 1045

## R

Real 1084  
reboot, YModule 811  
Recorded 300  
Reference 10, 1111  
registerAnglesCallback, YGyro 588  
RegisterDeviceArrivalCallback, YAPI 21  
RegisterDeviceRemovalCallback, YAPI 22  
RegisterHub, YAPI 23  
RegisterHubDiscoveryCallback, YAPI 24  
registerLogCallback, YModule 812  
RegisterLogFunction, YAPI 25  
registerQuaternionCallback, YGyro 589  
registerTimedReportCallback, YAccelerometer 64  
registerTimedReportCallback, YCarbonDioxide 141  
registerTimedReportCallback, YCompass 210  
registerTimedReportCallback, YCurrent 249  
registerTimedReportCallback, YGenericSensor 536  
registerTimedReportCallback, YGyro 590  
registerTimedReportCallback, YHumidity 654  
registerTimedReportCallback, YLightSensor 721  
registerTimedReportCallback, YMagnetometer 763  
registerTimedReportCallback, YPower 934  
registerTimedReportCallback, YPressure 974  
registerTimedReportCallback, YQt 1074  
registerTimedReportCallback, YSensor 1212  
registerTimedReportCallback, YTemperature 1287  
registerTimedReportCallback, YTilt 1327  
registerTimedReportCallback, YVoc 1366  
registerTimedReportCallback, YVoltage 1405  
registerValueCallback, YAccelerometer 65  
registerValueCallback, YAnButton 103  
registerValueCallback, YCarbonDioxide 142  
registerValueCallback, YColorLed 172  
registerValueCallback, YCompass 211  
registerValueCallback, YCurrent 250  
registerValueCallback, YDataLogger 283  
registerValueCallback, YDigitalIO 357  
registerValueCallback, YDisplay 405  
registerValueCallback, YDualPower 470  
registerValueCallback, YFiles 497  
registerValueCallback, YGenericSensor 537

registerValueCallback, YGyro 591  
registerValueCallback, YHubPort 620  
registerValueCallback, YHumidity 655  
registerValueCallback, YLed 684  
registerValueCallback, YLightSensor 722  
registerValueCallback, YMagnetometer 764  
registerValueCallback, YNetwork 861  
registerValueCallback, YOsControl 897  
registerValueCallback, YPower 935  
registerValueCallback, YPressure 975  
registerValueCallback, YPwmOutput 1011  
registerValueCallback, YPwmPowerSource 1040  
registerValueCallback, YQt 1075  
registerValueCallback, YRealTimeClock 1105  
registerValueCallback, YRefFrame 1139  
registerValueCallback, YRelay 1174  
registerValueCallback, YSensor 1213  
registerValueCallback, YServo 1247  
registerValueCallback, YTemperature 1288  
registerValueCallback, YTilt 1328  
registerValueCallback, YVoc 1367  
registerValueCallback, YVoltage 1406  
registerValueCallback, YVSource 1437  
registerValueCallback, YWakeUpMonitor 1466  
registerValueCallback, YWakeUpSchedule 1504  
registerValueCallback, YWatchdog 1546  
registerValueCallback, YWireless 1585  
Relay 1147  
remove, YFiles 498  
reset, YDisplayLayer 438  
reset, YPower 936  
resetAll, YDisplay 406  
resetCounter, YAnButton 104  
resetSleepCountDown, YWakeUpMonitor 1467  
resetWatchdog, YWatchdog 1547  
revertFromFlash, YModule 813  
rgbMove, YColorLed 173

## S

save3DCalibration, YRefFrame 1140  
saveSequence, YDisplay 407  
saveToFlash, YModule 814  
SelectArchitecture, YAPI 26  
selectColorPen, YDisplayLayer 439  
selectEraser, YDisplayLayer 440  
selectFont, YDisplayLayer 441  
selectGrayPen, YDisplayLayer 442  
Sensor 1183  
Sequence 290, 300, 312  
Servo 1222  
set\_adminPassword, YNetwork 862  
set\_analogCalibration, YAnButton 105  
set\_autoStart, YDataLogger 284  
set\_autoStart, YWatchdog 1548  
set\_beacon, YModule 815  
set\_bearing, YRefFrame 1141  
set\_bitDirection, YDigitalIO 358  
set\_bitOpenDrain, YDigitalIO 359  
set\_bitPolarity, YDigitalIO 360

set\_bitState, YDigitalIO 361  
set\_blinking, YLed 685  
set\_brightness, YDisplay 408  
set\_calibrationMax, YAnButton 106  
set\_calibrationMin, YAnButton 107  
set\_callbackCredentials, YNetwork 863  
set\_callbackEncoding, YNetwork 864  
set\_callbackMaxDelay, YNetwork 865  
set\_callbackMethod, YNetwork 866  
set\_callbackMinDelay, YNetwork 867  
set\_callbackUrl, YNetwork 868  
set\_discoverable, YNetwork 869  
set\_dutyCycle, YPwmOutput 1012  
set\_dutyCycleAtPowerOn, YPwmOutput 1013  
set\_enabled, YDisplay 409  
set\_enabled, YHubPort 621  
set\_enabled, YPwmOutput 1014  
set\_enabled,YServo 1248  
set\_enabledAtPowerOn, YPwmOutput 1015  
set\_enabledAtPowerOn,YServo 1249  
set\_frequency, YPwmOutput 1016  
set\_highestValue, YAccelerometer 66  
set\_highestValue, YCarbonDioxide 143  
set\_highestValue, YCompass 212  
set\_highestValue, YCurrent 251  
set\_highestValue, YGenericSensor 538  
set\_highestValue, YGyro 592  
set\_highestValue, YHumidity 656  
set\_highestValue, YLightSensor 723  
set\_highestValue, YMagnetometer 765  
set\_highestValue, YPower 937  
set\_highestValue, YPressure 976  
set\_highestValue, YQt 1076  
set\_highestValue, YSensor 1214  
set\_highestValue, YTemperature 1289  
set\_highestValue, YTilt 1329  
set\_highestValue, YVoc 1368  
set\_highestValue, YVoltage 1407  
set\_hours, YWakeUpSchedule 1505  
set\_hslColor, YColorLed 174  
set\_logFrequency, YAccelerometer 67  
set\_logFrequency, YCarbonDioxide 144  
set\_logFrequency, YCompass 213  
set\_logFrequency, YCurrent 252  
set\_logFrequency, YGenericSensor 539  
set\_logFrequency, YGyro 593  
set\_logFrequency, YHumidity 657  
set\_logFrequency, YLightSensor 724  
set\_logFrequency, YMagnetometer 766  
set\_logFrequency, YPower 938  
set\_logFrequency, YPressure 977  
set\_logFrequency, YQt 1077  
set\_logFrequency, YSensor 1215  
set\_logFrequency, YTemperature 1290  
set\_logFrequency, YTilt 1330  
set\_logFrequency, YVoc 1369  
set\_logFrequency, YVoltage 1408  
set\_logicalName, YAccelerometer 68  
set\_logicalName, YAnButton 108  
set\_logicalName, YCarbonDioxide 145  
set\_logicalName, YColorLed 175  
set\_logicalName, YCompass 214  
set\_logicalName, YCurrent 253  
set\_logicalName, YDataLogger 285  
set\_logicalName, YDigitalIO 362  
set\_logicalName, YDisplay 410  
set\_logicalName, YDualPower 471  
set\_logicalName, YFiles 499  
set\_logicalName, YGenericSensor 540  
set\_logicalName, YGyro 594  
set\_logicalName, YHubPort 622  
set\_logicalName, YHumidity 658  
set\_logicalName, YLed 686  
set\_logicalName, YLightSensor 725  
set\_logicalName, YMagnetometer 767  
set\_logicalName, YModule 816  
set\_logicalName, YNetwork 870  
set\_logicalName, YOsControl 898  
set\_logicalName, YPower 939  
set\_logicalName, YPressure 978  
set\_logicalName, YPwmOutput 1017  
set\_logicalName, YPwmPowerSource 1041  
set\_logicalName, YQt 1078  
set\_logicalName, YRealTimeClock 1106  
set\_logicalName, YRefFrame 1142  
set\_logicalName, YRelay 1175  
set\_logicalName, YSensor 1216  
set\_logicalName, YServo 1250  
set\_logicalName, YTemperature 1291  
set\_logicalName, YTilt 1331  
set\_logicalName, YVoc 1370  
set\_logicalName, YVoltage 1409  
set\_logicalName, YVSource 1438  
set\_logicalName, YWakeUpMonitor 1468  
set\_logicalName, YWakeUpSchedule 1506  
set\_logicalName, YWatchdog 1549  
set\_logicalName, YWireless 1586  
set\_lowestValue, YAccelerometer 69  
set\_lowestValue, YCarbonDioxide 146  
set\_lowestValue, YCompass 215  
set\_lowestValue, YCurrent 254  
set\_lowestValue, YGenericSensor 541  
set\_lowestValue, YGyro 595  
set\_lowestValue, YHumidity 659  
set\_lowestValue, YLightSensor 726  
set\_lowestValue, YMagnetometer 768  
set\_lowestValue, YPower 940  
set\_lowestValue, YPressure 979  
set\_lowestValue, YQt 1079  
set\_lowestValue, YSensor 1217  
set\_lowestValue, YTemperature 1292  
set\_lowestValue, YTilt 1332  
set\_lowestValue, YVoc 1371  
set\_lowestValue, YVoltage 1410  
set\_luminosity, YLed 687  
set\_luminosity, YModule 817  
set\_maxTimeOnStateA, YRelay 1176  
set\_maxTimeOnStateA, YWatchdog 1550

set\_maxTimeOnStateB, YRelay 1177  
set\_maxTimeOnStateB, YWatchdog 1551  
set\_minutes, YWakeUpSchedule 1507  
set\_minutesA, YWakeUpSchedule 1508  
set\_minutesB, YWakeUpSchedule 1509  
set\_monthDays, YWakeUpSchedule 1510  
set\_months, YWakeUpSchedule 1511  
set\_mountPosition, YRefFrame 1143  
set\_neutral, YServo 1251  
set\_nextWakeUp, YWakeUpMonitor 1469  
set\_orientation, YDisplay 411  
set\_output, YRelay 1178  
set\_output, YWatchdog 1552  
set\_outputVoltage, YDigitalIO 363  
set\_period, YPwmOutput 1018  
set\_portDirection, YDigitalIO 364  
set\_portOpenDrain, YDigitalIO 365  
set\_portPolarity, YDigitalIO 366  
set\_portState, YDigitalIO 367  
set\_position, YServo 1252  
set\_positionAtPowerOn, YServo 1253  
set\_power, YLed 688  
set\_powerControl, YDualPower 472  
set\_powerDuration, YWakeUpMonitor 1470  
set\_powerMode, YPwmPowerSource 1042  
set\_primaryDNS, YNetwork 871  
set\_pulseDuration, YPwmOutput 1019  
set\_range, YServo 1254  
set\_recording, YDataLogger 286  
set\_reportFrequency, YAccelerometer 70  
set\_reportFrequency, YCarbonDioxide 147  
set\_reportFrequency, YCompass 216  
set\_reportFrequency, YCurrent 255  
set\_reportFrequency, YGenericSensor 542  
set\_reportFrequency, YGyro 596  
set\_reportFrequency, YHumidity 660  
set\_reportFrequency, YLightSensor 727  
set\_reportFrequency, YMagnetometer 769  
set\_reportFrequency, YPower 941  
set\_reportFrequency, YPressure 980  
set\_reportFrequency, YQt 1080  
set\_reportFrequency, YSensor 1218  
set\_reportFrequency, YTemperature 1293  
set\_reportFrequency, YTilt 1333  
set\_reportFrequency, YVoc 1372  
set\_reportFrequency, YVoltage 1411  
set\_resolution, YAccelerometer 71  
set\_resolution, YCarbonDioxide 148  
set\_resolution, YCompass 217  
set\_resolution, YCurrent 256  
set\_resolution, YGenericSensor 543  
set\_resolution, YGyro 597  
set\_resolution, YHumidity 661  
set\_resolution, YLightSensor 728  
set\_resolution, YMagnetometer 770  
set\_resolution, YPower 942  
set\_resolution, YPressure 981  
set\_resolution, YQt 1081  
set\_resolution, YSensor 1219  
set\_resolution, YTemperature 1294  
set\_resolution, YTilt 1334  
set\_resolution, YVoc 1373  
set\_resolution, YVoltage 1412  
set\_rgbColor, YColorLed 176  
set\_rgbColorAtPowerOn, YColorLed 177  
set\_running, YWatchdog 1553  
set\_secondaryDNS, YNetwork 872  
set\_sensitivity, YAnButton 109  
set\_sensorType, YTemperature 1295  
set\_signalRange, YGenericSensor 544  
set\_sleepCountdown, YWakeUpMonitor 1471  
set\_startupSeq, YDisplay 412  
set\_state, YRelay 1179  
set\_state, YWatchdog 1554  
set\_stateAtPowerOn, YRelay 1180  
set\_stateAtPowerOn, YWatchdog 1555  
set\_timeUTC, YDataLogger 287  
set\_triggerDelay, YWatchdog 1556  
set\_triggerDuration, YWatchdog 1557  
set\_unit, YGenericSensor 545  
set\_unixTime, YRealTimeClock 1107  
set\_usbBandwidth, YModule 818  
set\_userData, YAccelerometer 72  
set\_userData, YAnButton 110  
set\_userData, YCarbonDioxide 149  
set\_userData, YColorLed 178  
set\_userData, YCompass 218  
set\_userData, YCurrent 257  
set\_userData, YDataLogger 288  
set\_userData, YDigitalIO 368  
set\_userData, YDisplay 413  
set\_userData, YDualPower 473  
set\_userData, YFiles 500  
set\_userData, YGenericSensor 546  
set\_userData, YGyro 598  
set\_userData, YHubPort 623  
set\_userData, YHumidity 662  
set\_userData, YLed 689  
set\_userData, YLightSensor 729  
set\_userData, YMagnetometer 771  
set\_userData, YModule 819  
set\_userData, YNetwork 873  
set\_userData, YOsControl 899  
set\_userData, YPower 943  
set\_userData, YPressure 982  
set\_userData, YPwmOutput 1020  
set\_userData, YPwmPowerSource 1043  
set\_userData, YQt 1082  
set\_userData, YRealTimeClock 1108  
set\_userData, YRefFrame 1144  
set\_userData, YRelay 1181  
set\_userData, YSensor 1220  
set\_userData, YServo 1255  
set\_userData, YTemperature 1296  
set\_userData, YTilt 1335  
set\_userData, YVoc 1374  
set\_userData, YVoltage 1413  
set\_userData, YVSource 1439

set(userData, YWakeUpMonitor 1472  
set(userData, YWakeUpSchedule 1512  
set(userData, YWatchdog 1558  
set(userData, YWireless 1587  
set(userPassword, YNetwork 874  
set\_utcOffset, YRealTimeClock 1109  
set\_valueInterval, YDataRun 298  
set\_valueRange, YGenericSensor 547  
set\_voltage, YVSource 1440  
set\_weekDays, YWakeUpSchedule 1513  
set\_wwwWatchdogDelay, YNetwork 875  
setAntialiasingMode, YDisplayLayer 443  
setConsoleBackground, YDisplayLayer 444  
setConsoleMargins, YDisplayLayer 445  
setConsoleWordWrap, YDisplayLayer 446  
setLayerPosition, YDisplayLayer 447  
shutdown, YOsControl 900  
Sleep, YAPI 27  
sleep, YWakeUpMonitor 1473  
sleepFor, YWakeUpMonitor 1474  
sleepUntil, YWakeUpMonitor 1475  
Source 3, 1415  
start3DCalibration, YRefFrame 1145  
stopSequence, YDisplay 414  
Supply 450  
swapLayerContent, YDisplay 415

## T

Temperature 1257  
Tilt 1298  
Time 1084  
toggle\_bitState, YDigitalIO 369  
triggerFirmwareUpdate, YModule 820  
TriggerHubDiscovery, YAPI 28

## U

Unformatted 312  
unhide, YDisplayLayer 448  
UnregisterHub, YAPI 29  
UpdateDeviceList, YAPI 30  
upload, YDisplay 416  
upload, YFiles 501  
useDHCP, YNetwork 876  
useStaticIP, YNetwork 877

## V

Value 773  
Voltage 1376, 1415  
voltageMove, YVSource 1441

## W

wakeUp, YWakeUpMonitor 1476  
WakeUpMonitor 1443  
WakeUpSchedule 1478  
Watchdog 1515  
Wireless 1560

## Y

yAccelerometer 34-72  
yAnButton 76-110  
yAPI 12-30  
yCarbonDioxide 114-149  
yCheckLogicalName 12  
yColorLed 152-178  
yCompass 182-218  
yCurrent 222-257  
yDataLogger 260-288  
yDataRun 290-298  
yDataSet 301-310  
yDataStream 313-325  
yDigitalIO 329-369  
yDisableExceptions 13  
yDisplay 373-416  
yDisplayLayer 419-448  
yDualPower 451-473  
yEnableExceptions 14  
yFiles 476-501  
yFindAccelerometer 34  
yFindAnButton 76  
yFindCarbonDioxide 114  
yFindColorLed 152  
yFindCompass 182  
yFindCurrent 222  
yFindDataLogger 260  
yFindDigitalIO 329  
yFindDisplay 373  
yFindDualPower 451  
yFindFiles 476  
yFindGenericSensor 505  
yFindGyro 551  
yFindHubPort 601  
yFindHumidity 627  
yFindLed 665  
yFindLightSensor 693  
yFindMagnetometer 733  
yFindModule 781  
yFindNetwork 825  
yFindOsControl 880  
yFindPower 904  
yFindPressure 947  
yFindPwmOutput 986  
yFindPwmPowerSource 1023  
yFindQt 1047  
yFindRealTimeClock 1085  
yFindRefFrame 1113  
yFindRelay 1149  
yFindSensor 1185  
yFindServo 1224  
yFindTemperature 1259  
yFindTilt 1300  
yFindVoc 1339  
yFindVoltage 1378  
yFindVSource 1416  
yFindWakeUpMonitor 1445  
yFindWakeUpSchedule 1480

yFindWatchdog 1517  
yFindWireless 1561  
yFirstAccelerometer 35  
yFirstAnButton 77  
yFirstCarbonDioxide 115  
yFirstColorLed 153  
yFirstCompass 183  
yFirstCurrent 223  
yFirstDataLogger 261  
yFirstDigitalIO 330  
yFirstDisplay 374  
yFirstDualPower 452  
yFirstFiles 477  
yFirstGenericSensor 506  
yFirstGyro 552  
yFirstHubPort 602  
yFirstHumidity 628  
yFirstLed 666  
yFirstLightSensor 694  
yFirstMagnetometer 734  
yFirstModule 782  
yFirstNetwork 826  
yFirstOsControl 881  
yFirstPower 905  
yFirstPressure 948  
yFirstPwmOutput 987  
yFirstPwmPowerSource 1024  
yFirstQt 1048  
yFirstRealTimeClock 1086  
yFirstRefFrame 1114  
yFirstRelay 1150  
yFirstSensor 1186  
yFirstServo 1225  
yFirstTemperature 1260  
yFirstTilt 1301  
yFirstVoc 1340  
yFirstVoltage 1379  
yFirstVSource 1417  
yFirstWakeUpMonitor 1446  
yFirstWakeUpSchedule 1481  
yFirstWatchdog 1518  
yFirstWireless 1562  
yFreeAPI 15  
YGenericSensor 505-547  
yGetAPIVersion 16  
yGetTickCount 17  
YGyro 551-598  
yHandleEvents 18  
YHubPort 601-623  
YHumidity 627-662  
yInitAPI 19  
YLed 665-689  
YLightSensor 693-729  
YMagnetometer 733-771  
YMeasure 773-777  
YModule 781-820  
YNetwork 825-877  
Yocto-Demo 3  
Yocto-hub 600  
YOsControl 880-900  
YPower 904-943  
yPreregisterHub 20  
YPressure 947-982  
YPwmOutput 986-1020  
YPwmPowerSource 1023-1043  
YQt 1047-1082  
YRealTimeClock 1085-1109  
YRefFrame 1113-1145  
yRegisterDeviceArrivalCallback 21  
yRegisterDeviceRemovalCallback 22  
yRegisterHub 23  
yRegisterHubDiscoveryCallback 24  
yRegisterLogFunction 25  
YRelay 1149-1181  
ySelectArchitecture 26  
YSensor 1185-1220  
YServo 1224-1255  
ySleep 27  
YTemperature 1259-1296  
YTilt 1300-1335  
yTriggerHubDiscovery 28  
yUnregisterHub 29  
yUpdateDeviceList 30  
YVoc 1339-1374  
YVoltage 1378-1413  
YVSource 1416-1441  
YWakeUpMonitor 1445-1476  
YWakeUpSchedule 1480-1513  
YWatchdog 1517-1558  
YWireless 1561-1587