



# 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. Altitude function interface .....	74
3.4. AnButton function interface .....	116
3.5. CarbonDioxide function interface .....	154
3.6. ColorLed function interface .....	193
3.7. Compass function interface .....	222
3.8. Current function interface .....	262
3.9. DataLogger function interface .....	301
3.10. Formatted data sequence .....	335
3.11. Recorded data sequence .....	345
3.12. Unformatted data sequence .....	357
3.13. Digital IO function interface .....	372
3.14. Display function interface .....	416
3.15. DisplayLayer object interface .....	463
3.16. External power supply control interface .....	495
3.17. Files function interface .....	520
3.18. GenericSensor function interface .....	548
3.19. Gyroscope function interface .....	597
3.20. Yocto-hub port interface .....	648
3.21. Humidity function interface .....	673
3.22. Led function interface .....	712
3.23. LightSensor function interface .....	739
3.24. Magnetometer function interface .....	781

3.25. Measured value .....	823
3.26. Module control interface .....	829
3.27. Motor function interface .....	876
3.28. Network function interface .....	917
3.29. OS control .....	974
3.30. Power function interface .....	997
3.31. Pressure function interface .....	1040
3.32. PwmInput function interface .....	1079
3.33. Pwm function interface .....	1127
3.34. PwmPowerSource function interface .....	1165
3.35. Quaternion interface .....	1188
3.36. Real Time Clock function interface .....	1227
3.37. Reference frame configuration .....	1254
3.38. Relay function interface .....	1290
3.39. Sensor function interface .....	1326
3.40. SerialPort function interface .....	1365
3.41. Servo function interface .....	1422
3.42. Temperature function interface .....	1457
3.43. Tilt function interface .....	1498
3.44. Voc function interface .....	1537
3.45. Voltage function interface .....	1576
3.46. Voltage source function interface .....	1615
3.47. WakeUpMonitor function interface .....	1643
3.48. WakeUpSchedule function interface .....	1678
3.49. Watchdog function interface .....	1715
3.50. Wireless function interface .....	1760

<b>Index .....</b>	<b>1791</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 object 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':
    # retrieve 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()

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

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 guaranties 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.

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

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

---

**YAPI**

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()**

**YAPI**

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

---

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()

YAPI

### yGetTickCount()YAPI.GetTickCount()

---

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() yHandleEvents()YAPI.HandleEvents()

YAPI

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()**

YAPI

**yInitAPI()YAPI.InitAPI()**

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() yRegisterHub()YAPI.RegisterHub()

YAPI

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**: that 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@address: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 not possible to reliably distinguish between a Hard Float (armhf) and a Soft Float (armel) install. For in 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. Possibles value 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()**  
**yTriggerHubDiscovery()YAPI.TriggerHubDiscovery()**

---

**YAPI**

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.

```
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()

YAPI

### yUnregisterHub()YAPI.UnregisterHub()

---

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

---

YAPI

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

### 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, in g, as a floating point number.

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

Returns the current value of the acceleration, in g, as a floating point number.

#### **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`.

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

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

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

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

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

Returns the logical name of the accelerometer.

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

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

**accelerometer**→**get\_module()**

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

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

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

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

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

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

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

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

Returns the resolution of the measured values.

**accelerometer**→**get\_unit()**

Returns the measuring unit for the acceleration.

**accelerometer**→**get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**accelerometer**→**get\_xValue()**

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

**accelerometer**→**get\_yValue()**

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

**accelerometer**→**get\_zValue()**

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

**accelerometer**→**isOnline()**

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

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

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

**accelerometer**→**load(msValidity)**

Preloads the accelerometer cache with a specified validity duration.

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

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

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

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

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

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

**accelerometer**→**registerTimedReportCallback(callback)**

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

**accelerometer**→**registerValueCallback(callback)**

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()**  
**yFirstAccelerometer()**  
**YAccelerometer.FirstAccelerometer()**

---

**YAccelerometer**

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` 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)=RELAYLO1-123456.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 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()`

`YAccelerometer`

`accelerometer`→`currentRawValue()`

`accelerometer.get_currentRawValue()`

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in g, as a floating point number.

```
def get_currentRawValue()
```

**Returns :**

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

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

---

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

---

Returns the current value of the acceleration, in g, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the acceleration, in g, as a floating point number

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

---

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()**

**YAccelerometer**

**accelerometer**→**friendlyName()**

**accelerometer.get\_friendlyName()**

---

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 example: `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()****YAccelerometer****accelerometer**→**functionDescriptor()****accelerometer.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`.

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

**YAccelerometer**

**accelerometer**→**functionId()**

**accelerometer.get\_functionId()**

---

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()****YAccelerometer****accelerometer**→**hardwareId()****accelerometer.get\_hardwareId()**

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the accelerometer (ex: `RELAYLO1-123456.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()****YAccelerometer****accelerometer**→**logFrequency()****accelerometer.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`.

**accelerometer→get\_logicalName()**

**YAccelerometer**

**accelerometer→logicalName()**

**accelerometer.get\_logicalName()**

---

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

---

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 on-line.

**Returns :**

an instance of YModule



---

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

**accelerometer→get\_reportFrequency()**

**YAccelerometer**

**accelerometer→reportFrequency()**

**accelerometer.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.

---

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

**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()****YAccelerometer****accelerometer→userData()****accelerometer.getUserData()**

---

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
def getUserData( )
```

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()****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()****YAccelerometer****accelerometer→setHighestValue()****accelerometer.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.

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

---

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()**

**YAccelerometer**

**accelerometer**→**setLowestValue()**

**accelerometer.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.

---

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

**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 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

### 3.3. Altitude 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_altitude.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YAltitude = yoctolib.YAltitude;
php	require_once('yocto_altitude.php');
c++	#include "yocto_altitude.h"
m	#import "yocto_altitude.h"
pas	uses yocto_altitude;
vb	yocto_altitude.vb
cs	yocto_altitude.cs
java	import com.yoctopuce.YoctoAPI.YAltitude;
py	from yocto_altitude import *

#### Global functions

##### yFindAltitude(func)

Retrieves an altimeter for a given identifier.

##### yFirstAltitude()

Starts the enumeration of altimeters currently accessible.

#### YAltitude methods

##### altitude→calibrateFromPoints(rawValues, refValues)

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

##### altitude→describe()

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

##### altitude→get\_advertisedValue()

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

##### altitude→get\_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in meters, as a floating point number.

##### altitude→get\_currentValue()

Returns the current value of the altitude, in meters, as a floating point number.

##### altitude→get\_errorMessage()

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

##### altitude→get\_errorType()

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

##### altitude→get\_friendlyName()

Returns a global identifier of the altimeter in the format `MODULE_NAME . FUNCTION_NAME`.

##### altitude→get\_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

##### altitude→get\_functionId()

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

##### altitude→get\_hardwareId()

Returns the unique hardware identifier of the altimeter in the form `SERIAL . FUNCTIONID`.

**altitude**→**get\_highestValue()**

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

**altitude**→**get\_logFrequency()**

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

**altitude**→**get\_logicalName()**

Returns the logical name of the altimeter.

**altitude**→**get\_lowestValue()**

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

**altitude**→**get\_module()**

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

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

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

**altitude**→**get\_qnh()**

Returns the barometric pressure adjusted to sea level used to compute the altitude (QNH).

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

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

**altitude**→**get\_reportFrequency()**

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

**altitude**→**get\_resolution()**

Returns the resolution of the measured values.

**altitude**→**get\_unit()**

Returns the measuring unit for the altitude.

**altitude**→**get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**altitude**→**isOnline()**

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

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

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

**altitude**→**load(msValidity)**

Preloads the altimeter cache with a specified validity duration.

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

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

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

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

**altitude**→**nextAltitude()**

Continues the enumeration of altimeters started using `yFirstAltitude()`.

**altitude**→**registerTimedReportCallback(callback)**

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

**altitude**→**registerValueCallback(callback)**

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

**altitude**→**set\_currentValue(newval)**

Changes the current estimated altitude.

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

Changes the recorded maximal value observed.

### 3. Reference

---

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

Changes the datalogger recording frequency for this function.

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

Changes the logical name of the altimeter.

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

Changes the recorded minimal value observed.

**altitude**→**set\_qnh**(**newval**)

Changes the barometric pressure adjusted to sea level used to compute the altitude (QNH).

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

Changes the timed value notification frequency for this function.

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

Changes the resolution of the measured physical values.

**altitude**→**set\_userData**(**data**)

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

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

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



## YAltitude.FindAltitude() yFindAltitude()YAltitude.FindAltitude()

YAltitude

Retrieves an altimeter for a given identifier.

```
def FindAltitude( 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 altimeter is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAltitude.isOnline()` to test if the altimeter is indeed online at a given time. In case of ambiguity when looking for an altimeter 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 altimeter

**Returns :**

a `YAltitude` object allowing you to drive the altimeter.

## **YAltitude.FirstAltitude() yFirstAltitude()YAltitude.FirstAltitude()**

---

**YAltitude**

Starts the enumeration of altimeters currently accessible.

```
def FirstAltitude( )
```

Use the method `YAltitude.nextAltitude()` to iterate on next altimeters.

**Returns :**

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

---

**altitude**→**calibrateFromPoints()**  
**altitude.calibrateFromPoints()****YAltitude**

---

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.

**altitude**→**describe()****altitude.describe()****YAltitude**

Returns a short text that describes unambiguously the instance of the altimeter 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(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 altimeter (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

---

**altitude**→**get\_advertisedValue()**

**YAltitude**

**altitude**→**advertisedValue()**

**altitude.get\_advertisedValue()**

---

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

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the altimeter (no more than 6 characters).

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

**altitude**→**get\_currentRawValue()**

**YAltitude**

**altitude**→**currentRawValue()**

**altitude.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in meters, as a floating point number.

```
def get_currentRawValue()
```

**Returns :**

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

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

---

**altitude**→**get\_currentValue()****YAltitude****altitude**→**currentValue()****altitude.get\_currentValue()**

---

Returns the current value of the altitude, in meters, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the altitude, in meters, as a floating point number

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

**altitude**→**get\_errorMessage()**

**YAltitude**

**altitude**→**errorMessage()****altitude.errorMessage()**

---

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

```
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 altimeter object



---

**altitude**→**get\_errorType()****YAltitude****altitude**→**errorType()****altitude.get\_errorType()**

---

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

```
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 altimeter object

**altitude**→**get\_friendlyName()**

**YAltitude**

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

---

Returns a global identifier of the altimeter in the format `MODULE_NAME . FUNCTION_NAME`.

```
def get_friendlyName( )
```

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

**Returns :**

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

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

---

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

---

**YAltitude**

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`.

**altitude**→**get\_functionId()**

**YAltitude**

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

---

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

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the altimeter (ex: `relay1`)

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

---

**altitude**→**get\_hardwareId()****YAltitude****altitude**→**hardwareId()****altitude.get\_hardwareId()**

---

Returns the unique hardware identifier of the altimeter 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 altimeter (for example `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the altimeter (ex: `RELAYLO1-123456.relay1`)

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

**altitude**→**get\_highestValue()**

**YAltitude**

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

---

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

```
def get_highestValue( )
```

**Returns :**

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

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

---

**altitude**→**get\_logFrequency()****YAltitude****altitude**→**logFrequency()****altitude.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`.

**altitude**→**get\_logicalName()**

**YAltitude**

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

---

Returns the logical name of the altimeter.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the altimeter.

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



---

**altitude**→**get\_lowestValue()****YAltitude****altitude**→**lowestValue()****altitude.get\_lowestValue()**

---

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

```
def get_lowestValue()
```

**Returns :**

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

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

**altitude**→**get\_module()**

**YAltitude**

**altitude**→**module()****altitude.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

---

**altitude**→**get\_qnh()****YAltitude****altitude**→**qnh()****altitude.get\_qnh()**

---

Returns the barometric pressure adjusted to sea level used to compute the altitude (QNH).

```
def get_qnh( )
```

**Returns :**

a floating point number corresponding to the barometric pressure adjusted to sea level used to compute the altitude (QNH)

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

**altitude**→**get\_recordedData()****YAltitude****altitude**→**recordedData()****altitude.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.

---

**altitude**→**get\_reportFrequency()****YAltitude****altitude**→**reportFrequency()****altitude.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`.

**altitude**→**get\_resolution()**

**YAltitude**

**altitude**→**resolution()****altitude.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`.

---

**altitude**→**get\_unit()****YAltitude****altitude**→**unit()****altitude.get\_unit()**

---

Returns the measuring unit for the altitude.

```
def get_unit( )
```

**Returns :**

a string corresponding to the measuring unit for the altitude

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

**altitude**→**get\_userData()**

**YAltitude**

**altitude**→**userData()****altitude.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.



---

**altitude**→**isOnline()****altitude.isOnline()****YAltitude**

---

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

```
def isOnline( )
```

If there is a cached value for the altimeter 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 altimeter.

**Returns :**

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

**altitude**→**load()****altitude.load()****YAltitude**

Preloads the altimeter 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.

---

**altitude**→**loadCalibrationPoints()**  
**altitude.loadCalibrationPoints()**

---

**YAltitude**

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.

**altitude**→**nextAltitude()****altitude.nextAltitude()**

**YAltitude**

---

Continues the enumeration of altimeters started using `yFirstAltitude()`.

```
def nextAltitude( )
```

**Returns :**

a pointer to a `YAltitude` object, corresponding to an altimeter currently online, or a `null` pointer if there are no more altimeters to enumerate.

---

**altitude**→**registerTimedReportCallback()**  
**altitude.registerTimedReportCallback()**

---

**YAltitude**

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.

**altitude**→**registerValueCallback()**  
**altitude.registerValueCallback()****YAltitude**

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.

---

**altitude**→**set\_currentValue()**  
**altitude**→**setCurrentValue()**  
**altitude.set\_currentValue()**

---

**YAltitude**

Changes the current estimated altitude.

```
def set_currentValue( newval)
```

This allows to compensate for ambient pressure variations and to work in relative mode.

**Parameters :**

**newval** a floating point number corresponding to the current estimated altitude

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

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

---

**YAltitude**

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.



---

**altitude**→**set\_logFrequency()****YAltitude****altitude**→**setLogFrequency()****altitude.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.

**altitude**→**set\_logicalName()**

**YAltitude**

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

---

Changes the logical name of the altimeter.

```
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 altimeter.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**altitude**→**set\_lowestValue()****YAltitude****altitude**→**setLowestValue()****altitude.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.

**altitude**→**set\_qnh()**

**YAltitude**

**altitude**→**setQnh()****altitude.set\_qnh()**

---

Changes the barometric pressure adjusted to sea level used to compute the altitude (QNH).

```
def set_qnh( newval)
```

This enables you to compensate for atmospheric pressure changes due to weather conditions.

**Parameters :**

**newval** a floating point number corresponding to the barometric pressure adjusted to sea level used to compute the altitude (QNH)

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

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

---

**YAltitude**

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.

**altitude**→**set\_resolution()**

**YAltitude**

**altitude**→**setResolution()****altitude.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.

---

**altitude**→**set\_userdata()****YAltitude****altitude**→**setUserData()****altitude.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. 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 use 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	<code>&lt;script type='text/javascript' src='yocto_anbutton.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib'); var YAnButton = yoctolib.YAnButton;</code>
php	<code>require_once('yocto_anbutton.php');</code>
cpp	<code>#include "yocto_anbutton.h"</code>
m	<code>#import "yocto_anbutton.h"</code>
pas	<code>uses yocto_anbutton;</code>
vb	<code>yocto_anbutton.vb</code>
cs	<code>yocto_anbutton.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YAnButton;</code>
py	<code>from yocto_anbutton import *</code>

### 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 transitioned 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 transitioned 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() yFindAnButton()YAnButton.FindAnButton()

YAnButton

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)=RELAYLO1-123456.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)=RELAYLO1-123456.relay1`)

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

**YAnButton**

**anbutton**→**advertisedValue()**

**anbutton.get\_advertisedValue()**

---

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

---

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

---

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

---

**YAnButton**

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

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

---

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 example: `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`.

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

---

**YAnButton**

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the analog input (ex: `RELAYLO1-123456.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 transitioned 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 transitioned from open to closed)

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

---

**anbutton**→**get\_lastTimeReleased()****YAnButton****anbutton**→**lastTimeReleased()****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 transitioned 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 transitioned from closed to open)

On failure, throws an exception or returns `Y_LASTTIMERELASED_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 on-line.

**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_PULSECOUNTER_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()****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()****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

## 3.5. 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');
c++	#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, in ppm (vol), as a floating point number.

#### carbondioxide→get\_currentValue()

Returns the current value of the CO2 concentration, in ppm (vol), as a floating point number.

#### 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)`**

### 3. Reference

---

Changes the logical name of the CO2 sensor.

**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()  
yFirstCarbonDioxide()  
YCarbonDioxide.FirstCarbonDioxide()**

---

**YCarbonDioxide**

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()carbondioxide.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` 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)=RELAYLO1-123456.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 CO2 sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

---

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

---

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, in ppm (vol), as a floating point number.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number

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

---

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

---

Returns the current value of the CO2 concentration, in ppm (vol), as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the CO2 concentration, in ppm (vol), as a floating point number

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 example: `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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the CO2 sensor (ex: `RELAYLO1-123456.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()****YCarbonDioxide****carbondioxide→logFrequency()****carbondioxide.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.

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

---

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

```
def get_lowestValue( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the CO2 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 on-line.

**Returns :**

an instance of YModule

---

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

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

**YCarbonDioxide**

---

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()****YCarbonDioxide****carbondioxide**→**resolution()****carbondioxide.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`.

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()****YCarbonDioxide****carbondioxide**→**userData()****carbondioxide.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.

**carbondioxide**→**isOnline()****carbondioxide.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()carbondioxide.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.

**carbondioxide→loadCalibrationPoints()  
carbondioxide.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.

**carbondioxide**→**registerTimedReportCallback()**  
**carbondioxide.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.

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

## 3.6. 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	<code>&lt;script type='text/javascript' src='yocto_colorled.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib'); var YColorLed = yoctolib.YColorLed;</code>
php	<code>require_once('yocto_colorled.php');</code>
c++	<code>#include "yocto_colorled.h"</code>
m	<code>#import "yocto_colorled.h"</code>
pas	<code>uses yocto_colorled;</code>
vb	<code>yocto_colorled.vb</code>
cs	<code>yocto_colorled.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YColorLed;</code>
py	<code>from yocto_colorled import *</code>

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

---

**YColorLed**

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)=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 RGB led (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

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

**YColorLed**

**colorled**→**advertisedValue()**

**colorled.get\_advertisedValue()**

---

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

---

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 example: `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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the RGB led (ex: `RELAYLO1-123456.relay1`)

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



---

**colorled**→**get\_hslColor()****YColorLed****colorled**→**hslColor()****colorled.get\_hslColor()**

---

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 on-line.

**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**→**hslMove()****colorled.hslMove()****YColorLed**

---

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

```
def hslMove( 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: 0xHHSSL.

**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()`

**YColorLed**

`colorled`→`setLogicalName()`

`colorled.set_logicalName()`

---

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.

**colored**→**set\_rgbColorAtPowerOn()**  
**colored**→**setRgbColorAtPowerOn()**  
**colored.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( 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. 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	<code>&lt;script type='text/javascript' src='yocto_compass.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib'); var YCompass = yoctolib.YCompass;</code>
php	<code>require_once('yocto_compass.php');</code>
c++	<code>#include "yocto_compass.h"</code>
m	<code>#import "yocto_compass.h"</code>
pas	<code>uses yocto_compass;</code>
vb	<code>yocto_compass.vb</code>
cs	<code>yocto_compass.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YCompass;</code>
py	<code>from yocto_compass import *</code>

### 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, in degrees, as a floating point number.

#### **compass→get\_currentValue()**

Returns the current value of the relative bearing, in degrees, as a floating point number.

#### **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(data)**

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() yFirstCompass()YCompass.FirstCompass()**

---

**YCompass**

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` 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)=RELAYLO1-123456.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 compass (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)



---

**compass**→**get\_advertisedValue()****YCompass****compass**→**advertisedValue()****compass.get\_advertisedValue()**

---

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()**

**YCompass**

**compass**→**currentRawValue()**

**compass.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number.

```
def get_currentRawValue()
```

**Returns :**

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

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

---

**compass**→**get\_currentValue()****YCompass****compass**→**currentValue()****compass.get\_currentValue()**

---

Returns the current value of the relative bearing, in degrees, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the relative bearing, in degrees, as a floating point number

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

**compass**→**get\_errorMessage()**

**YCompass**

**compass**→**errorMessage()**

**compass.get\_errorMessage()**

---

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 example: `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()****YCompass****compass**→**functionDescriptor()****compass.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`.

**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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the compass (ex: `RELAYLO1-123456.relay1`)

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

**compass**→**get\_highestValue()**

**YCompass**

**compass**→**highestValue()**

**compass.get\_highestValue()**

---

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()****YCompass****compass**→**logFrequency()****compass.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`.

**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()**

**YCompass**

**compass**→**magneticHeading()**

**compass.get\_magneticHeading()**

---

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 on-line.

**Returns :**

an instance of `YModule`

**compass**→**get\_recordedData()****YCompass****compass**→**recordedData()****compass.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.



---

**compass**→**get\_reportFrequency()****YCompass****compass**→**reportFrequency()****compass.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`.

**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

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()****YCompass****compass**→**setHighestValue()****compass.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.

**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()****YCompass****compass**→**setLogicalName()****compass.set\_logicalName()**

---

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()****YCompass****compass**→**setReportFrequency()****compass.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.

**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

## 3.8. 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');
c++	#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, in mA, as a floating point number.

#### current→get\_currentValue()

Returns the current value of the current, in mA, as a floating point number.

#### 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 since the device was started.

**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 since the device was started.

**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.

**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.

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

Changes the timed value notification frequency for this function.

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

Changes the resolution of the measured physical 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() yFindCurrent()YCurrent.FindCurrent()

YCurrent

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` 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)=RELAYLO1-123456.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 current sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)



---

**current**→**get\_advertisedValue()****YCurrent****current**→**advertisedValue()****current.get\_advertisedValue()**

---

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()**

**YCurrent**

**current**→**currentRawValue()**

**current.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in mA, as a floating point number.

```
def get_currentRawValue()
```

**Returns :**

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

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

---

**current**→**get\_currentValue()****YCurrent****current**→**currentValue()****current.get\_currentValue()**

---

Returns the current value of the current, in mA, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the current, in mA, as a floating point number

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

**current**→**get\_errorMessage()**

**YCurrent**

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

---

Returns the error message of the latest error with the current 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 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 example: `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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the current sensor (ex: `RELAYLO1-123456.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 since the device was started.

```
def get_highestValue()
```

**Returns :**

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

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 since the device was started.

```
def get_lowestValue() ( )
```

**Returns :**

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

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 on-line.

**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()**

**YCurrent**

**current**→**reportFrequency()**

**current.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.



---

**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.

```
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.

---

**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.

```
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.

**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 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.

**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( 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. 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>
nodejs	var yoctolib = require('yoctolib'); var YDataLogger = yoctolib.YDataLogger;
php	require_once('yocto_datalogger.php');
c++	#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\_beaconDriven()

Return true if the data logger is synchronised with the localization beacon.

#### 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_beaconDriven(newval)`**

Changes the type of synchronisation of the data logger.

**`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)=RELAYLO1-123456.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)=RELAYLO1-123456.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()**

**YDataLogger**

**datalogger**→**advertisedValue()**

**datalogger.get\_advertisedValue()**

---

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\_beaconDriven()**

**YDataLogger**

**datalogger**→**beaconDriven()**

**datalogger.get\_beaconDriven()**

---

Return true if the data logger is synchronised with the localization beacon.

```
def get_beaconDriven( )
```

**Returns :**

either `Y_BEACONDRIVEN_OFF` or `Y_BEACONDRIVEN_ON`

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

---

**datalogger**→**get\_currentRunIndex()****YDataLogger****datalogger**→**currentRunIndex()****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.

```
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()**

**YDataLogger**

**datalogger**→**errorMessage()**

**datalogger.get\_errorMessage()**

---

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()**

**YDataLogger**

**datalogger**→**friendlyName()**

**datalogger.get\_friendlyName()**

---

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 example: `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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the data logger (ex: `RELAYLO1-123456.relay1`)

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

**datalogger**→**get\_logicalName()**

**YDataLogger**

**datalogger**→**logicalName()**

**datalogger.get\_logicalName()**

---

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 on-line.

**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\_beaconDriven()**  
**datalogger**→**setBeaconDriven()**  
**datalogger.set\_beaconDriven()**

---

**YDataLogger**

Changes the type of synchronisation of the data logger.

```
def set_beaconDriven( newval)
```

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

**Parameters :**

**newval** either `Y_BEACONDRIVEN_OFF` or `Y_BEACONDRIVEN_ON`, according to the type of synchronisation of the data logger

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

**datalogger**→**set\_logicalName()****YDataLogger****datalogger**→**setLogicalName()****datalogger.set\_logicalName()**

---

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()****YDataLogger****datalogger**→**setRecording()****datalogger.set\_recording()**

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

## 3.10. 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');
c++	#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	
<b>datarun</b> → <b>get_averageValue(measureName, pos)</b>	Returns the average value of the measure observed at the specified time period.
<b>datarun</b> → <b>get_duration()</b>	Returns the duration (in seconds) of the data run.
<b>datarun</b> → <b>get_maxValue(measureName, pos)</b>	Returns the maximal value of the measure observed at the specified time period.
<b>datarun</b> → <b>get_measureNames()</b>	Returns the names of the measures recorded by the data logger.
<b>datarun</b> → <b>get_minValue(measureName, pos)</b>	Returns the minimal value of the measure observed at the specified time period.
<b>datarun</b> → <b>get_startTimeUTC()</b>	Returns the start time of the data run, relative to the Jan 1, 1970.
<b>datarun</b> → <b>get_valueCount()</b>	Returns the number of values accessible in this run, given the selected data samples interval.
<b>datarun</b> → <b>get_valueInterval()</b>	Returns the number of seconds covered by each value in this run.
<b>datarun</b> → <b>set_valueInterval(valueInterval)</b>	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()****YDataRun****datarun**→**measureNames()****datarun.get\_measureNames()**

---

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()`

`YDataRun`

`datarun`→`setValueInterval()`

`datarun.set_valueInterval()`

---

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

## 3.11. 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 instantiated 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	<code>&lt;script type='text/javascript' src='yocto_api.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib'); var YAPI = yoctolib.YAPI; var YModule = yoctolib.YModule;</code>
php	<code>require_once('yocto_api.php');</code>
c++	<code>#include "yocto_api.h"</code>
m	<code>#import "yocto_api.h"</code>
pas	<code>uses yocto_api;</code>
vb	<code>yocto_api.vb</code>
cs	<code>yocto_api.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
py	<code>from yocto_api import *</code>

### 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 be 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.

### 3. Reference

---

#### **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 instantiated 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**→**get\_startTimeUTC()****YDataSet****dataset**→**startTimeUTC()****dataset.get\_startTimeUTC()**

---

Returns the start time of the dataset, relative to the Jan 1, 1970.

```
def get_startTimeUTC( )
```

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.



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

YDataStream methods	
<b>datastream</b> → <a href="#">get_averageValue()</a>	Returns the average of all measures observed within this stream.
<b>datastream</b> → <a href="#">get_columnCount()</a>	Returns the number of data columns present in this stream.
<b>datastream</b> → <a href="#">get_columnNames()</a>	Returns the title (or meaning) of each data column present in this stream.
<b>datastream</b> → <a href="#">get_data(row, col)</a>	Returns a single measure from the data stream, specified by its row and column index.
<b>datastream</b> → <a href="#">get_dataRows()</a>	Returns the whole data set contained in the stream, as a bidimensional table of numbers.
<b>datastream</b> → <a href="#">get_dataSamplesIntervalMs()</a>	Returns the number of milliseconds between two consecutive rows of this data stream.
<b>datastream</b> → <a href="#">get_duration()</a>	Returns the approximate duration of this stream, in seconds.
<b>datastream</b> → <a href="#">get_maxValue()</a>	Returns the largest measure observed within this stream.
<b>datastream</b> → <a href="#">get_minValue()</a>	Returns the smallest measure observed within this stream.
<b>datastream</b> → <a href="#">get_rowCount()</a>	Returns the number of data rows present in this stream.
<b>datastream</b> → <a href="#">get_runIndex()</a>	Returns the run index of the data stream.
<b>datastream</b> → <a href="#">get_startTime()</a>	Returns the relative start time of the data stream, measured in seconds.
<b>datastream</b> → <a href="#">get_startTimeUTC()</a>	

### 3. Reference

---

Returns the start time of the data stream, relative to the Jan 1, 1970.

---

**datastream**→**get\_averageValue()****YDataStream****datastream**→**averageValue()****datastream.get\_averageValue()**

---

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()**

**YDataStream**

**datastream→columnCount()**

**datastream.get\_columnCount()**

---

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()****YDataStream****datastream**→**columnNames()****datastream.get\_columnNames()**

---

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()**

**YDataStream**

**datastream**→**dataSamplesIntervalMs()**

**datastream.get\_dataSamplesIntervalMs()**

---

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**→**get\_rowCount()**

**YDataStream**

**datastream**→**rowCount()****datastream.get\_rowCount()**

---

Returns the number of data rows present in this stream.

```
def get_rowCount( )
```

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**→**get\_startTime()**

**YDataStream**

**datastream**→**startTime()****datastream.get\_startTime()**

---

Returns the relative start time of the data stream, measured in seconds.

```
def get_startTime( )
```

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 `get_startTimeUTC()`.

**Returns :**

an unsigned number corresponding to the number of seconds between the start of the run and the beginning of this data stream.

---

**datastream**→**get\_startTimeUTC()****YDataStream****datastream**→**startTimeUTC()****datastream.get\_startTimeUTC()**

---

Returns the start time of the data stream, relative to the Jan 1, 1970.

```
def get_startTimeUTC( )
```

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).

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

### 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`.



**digitalio**→**get\_functionDescriptor()**

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

**digitalio**→**get\_functionId()**

Returns the hardware identifier of the digital IO port, without reference to the module.

**digitalio**→**get\_hardwareId()**

Returns the unique hardware identifier of the digital IO port in the form SERIAL.FUNCTIONID.

**digitalio**→**get\_logicalName()**

Returns the logical name of the digital IO port.

**digitalio**→**get\_module()**

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

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

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

**digitalio**→**get\_outputVoltage()**

Returns the voltage source used to drive output bits.

**digitalio**→**get\_portDirection()**

Returns the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output.

**digitalio**→**get\_portOpenDrain()**

Returns the electrical interface for each bit of the port.

**digitalio**→**get\_portPolarity()**

Returns the polarity of all the bits of the port.

**digitalio**→**get\_portSize()**

Returns the number of bits implemented in the I/O port.

**digitalio**→**get\_portState()**

Returns the digital IO port state: bit 0 represents input 0, and so on.

**digitalio**→**get\_userData()**

Returns the value of the userData attribute, as previously stored using method set\_userData.

**digitalio**→**isOnline()**

Checks if the digital IO port is currently reachable, without raising any error.

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

Checks if the digital IO port is currently reachable, without raising any error (asynchronous version).

**digitalio**→**load(msValidity)**

Preloads the digital IO port cache with a specified validity duration.

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

Preloads the digital IO port cache with a specified validity duration (asynchronous version).

**digitalio**→**nextDigitalIO()**

Continues the enumeration of digital IO ports started using yFirstDigitalIO().

**digitalio**→**pulse(bitno, ms\_duration)**

Triggers a pulse on a single bit for a specified duration.

**digitalio**→**registerValueCallback(callback)**

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

**digitalio**→**set\_bitDirection(bitno, bitdirection)**

Changes the direction of a single bit from the I/O port.

**digitalio**→**set\_bitOpenDrain(bitno, opendrain)**

Changes the electrical interface of a single bit from the I/O port.

**digitalio**→**set\_bitPolarity(bitno, bitpolarity)**

### 3. Reference

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() yFirstDigitalIO()YDigitalIO.FirstDigitalIO()

---

YDigitalIO

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` 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)=RELAYLO1-123456.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)=RELAYLO1-123456.relay1`)

---

**digitalio**→**get\_advertisedValue()****YDigitalIO****digitalio**→**advertisedValue()****digitalio.get\_advertisedValue()**

---

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()**

**YDigitalIO**

**digitalio**→**errorMessage()**

**digitalio.get\_errorMessage()**

---

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 example: `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()****YDigitalIO****digitalio**→**functionDescriptor()****digitalio.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`.

**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_hardwareId()`**YDigitalIO****digitalio**→`hardwareId()`**digitalio**.`get_hardwareId()`

---

Returns the unique hardware identifier of the digital IO 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 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()****YDigitalIO****digitalio**→**module()****digitalio.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`

**digitalio**→**get\_outputVoltage()**

**YDigitalIO**

**digitalio**→**outputVoltage()**

**digitalio.get\_outputVoltage()**

---

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

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.

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()**

YDigitalIO

**digitalio**→**setBitOpenDrain()****digitalio.set\_bitOpenDrain()**

---

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()**

YDigitalIO

**digitalio**→**setBitState()****digitalio.set\_bitState()**

---

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()**

**YDigitalIO**

**digitalio**→**setLogicalName()**

**digitalio.set\_logicalName()**

---

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()****YDigitalIO****digitalio**→**setOutputVoltage()****digitalio.set\_outputVoltage()**

---

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( 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

---

**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.

## 3.14. 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>
nodejs	var yoctolib = require('yoctolib'); var YDisplay = yoctolib.YDisplay;
php	require_once('yocto_display.php');
c++	#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.

**display**→**getErrorType()**

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

**display**→**getFriendlyName()**

Returns a global identifier of the display in the format `MODULE_NAME . FUNCTION_NAME`.

**display**→**getFunctionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**display**→**getFunctionId()**

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

**display**→**getHardwareId()**

Returns the unique hardware identifier of the display in the form `SERIAL . FUNCTIONID`.

**display**→**getLayerCount()**

Returns the number of available layers to draw on.

**display**→**getLayerHeight()**

Returns the height of the layers to draw on, in pixels.

**display**→**getLayerWidth()**

Returns the width of the layers to draw on, in pixels.

**display**→**getLogicalName()**

Returns the logical name of the display.

**display**→**getModule()**

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

**display**→**getModule\_async(callback, context)**

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

**display**→**getOrientation()**

Returns the currently selected display orientation.

**display**→**getStartupSeq()**

Returns the name of the sequence to play when the displayed is powered on.

**display**→**getUserData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userdata`.

**display**→**isOnline()**

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

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

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

**display**→**load(msValidity)**

Preloads the display cache with a specified validity duration.

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

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

**display**→**newSequence()**

Starts to record all display commands into a sequence, for later replay.

**display**→**nextDisplay()**

Continues the enumeration of displays started using `yFirstDisplay()`.

**display**→**pauseSequence(delay\_ms)**

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

**display**→**playSequence(sequenceName)**

Replays a display sequence previously recorded using `newSequence()` and `saveSequence()`.

**display**→**registerValueCallback(callback)**

### 3. Reference

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(data)**

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() yFirstDisplay()YDisplay.FirstDisplay()

---

YDisplay

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)=RELAYLO1-123456.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)=RELAYLO1-123456.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()**

**YDisplay**

**display**→**advertisedValue()**

**display.get\_advertisedValue()**

---

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**.**get\_errorMessage()**

---

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

```
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 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 example: `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()****YDisplay****display**→**functionId()****display.get\_functionId()**

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the display (ex: `RELAYLO1-123456.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()**

**YDisplay**

**display**→**layerHeight()****display.get\_layerHeight()**

---

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`.

---

**display**→**get\_module()****YDisplay****display**→**module()****display.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`

**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( )
```

Using this function in a sequence will kill the sequence play-back. Don't use that function to reset the display at sequence start-up.

**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()**

**YDisplay**

**display**→**setBrightness()****display.set\_brightness()**

---

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()****YDisplay****display**→**setOrientation()****display.set\_orientation()**

---

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( 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

## `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.

## 3.15. 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;
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 *

### YDisplayLayer methods

#### displaylayer→clear()

Erases the whole content of the layer (makes it fully transparent).

#### displaylayer→clearConsole()

Blanks 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, bgcolor)

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.

### 3. Reference

**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 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**

Blanks 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()****YDisplayLayer****displaylayer**→**displayHeight()****displaylayer.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.

**displaylayer→get\_displayWidth()**

**YDisplayLayer**

**displaylayer→displayWidth()**

**displaylayer.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.

---

**displaylayer**→**get\_layerHeight()****YDisplayLayer****displaylayer**→**layerHeight()****displaylayer.get\_layerHeight()**

---

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()**

**YDisplayLayer**

**displaylayer→layerWidth()**

**displaylayer.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.

---

**displaylayer→hide()displaylayer.hide()**

---

**YDisplayLayer**

Hides the layer.

```
def hide( )
```

The state of the layer is perserved 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 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 lightest 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 fuzzyness 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.

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

### 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()**

### 3. Reference

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()

YDualPower

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()**

**YDualPower**

**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` 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)=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()**

**YDualPower**

**dualpower**→**advertisedValue()**

**dualpower.get\_advertisedValue()**

---

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()****YDualPower****dualpower**→**errorMessage()****dualpower.get\_errorMessage()**

---

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 example: `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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the power control (ex: `RELAYLO1-123456.relay1`)

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

**dualpower→get\_logicalName()**

**YDualPower**

**dualpower→logicalName()**

**dualpower.get\_logicalName()**

---

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 on-line.

**Returns :**

an instance of `YModule`

**dualpower**→**get\_powerControl()**

**YDualPower**

**dualpower**→**powerControl()**

**dualpower.get\_powerControl()**

---

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()****YDualPower****dualpower**→**powerState()****dualpower.get\_powerState()**

---

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()****YDualPower****dualpower**→**setLogicalName()****dualpower.set\_logicalName()**

---

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

## 3.17. 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');
c++	#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()

Reinitialize 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()

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(data)**

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() yFindFiles()YFiles.FindFiles()

YFiles

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)=RELAYLO1-123456.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)=RELAYLO1-123456.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

---

Reinitialize 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()****YFiles****files**→**errorType()****files.get\_errorType()**

---

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 example: `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`.



---

**files**→**get\_functionDescriptor()****YFiles****files**→**functionDescriptor()****files.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`.

**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`.

---

**files**→`get_hardwareId()`**YFiles****files**→`hardwareId()`**files.get\_hardwareId()**

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the filesystem (ex: `RELAYLO1-123456.relay1`)

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

**files**→**get\_list()**

**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 on-line.

**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.

---

**files**→**set\_logicalName()****YFiles****files**→**setLogicalName()****files.set\_logicalName()**

---

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( 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

---

**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.

## 3.18. 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_genericsensor.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YGenericSensor = yoctolib.YGenericSensor;
php	require_once('yocto_genericsensor.php');
cpp	#include "yocto_genericsensor.h"
m	#import "yocto_genericsensor.h"
pas	uses yocto_genericsensor;
vb	yocto_genericsensor.vb
cs	yocto_genericsensor.cs
java	import com.yoctopuce.YoctoAPI.YGenericSensor;
py	from yocto_genericsensor import *

### Global functions

#### yFindGenericSensor(func)

Retrieves a generic sensor for a given identifier.

#### yFirstGenericSensor()

Starts the enumeration of generic sensors currently accessible.

### YGenericSensor methods

#### genericsensor→calibrateFromPoints(rawValues, refValues)

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

#### genericsensor→describe()

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

#### genericsensor→get\_advertisedValue()

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

#### genericsensor→get\_currentRawValue()

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

#### genericsensor→get\_currentValue()

Returns the current measured value.

#### genericsensor→get\_errorMessage()

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

#### genericsensor→get\_errorType()

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

#### genericsensor→get\_friendlyName()

Returns a global identifier of the generic sensor in the format `MODULE_NAME . FUNCTION_NAME`.

#### genericsensor→get\_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

#### genericsensor→get\_functionId()

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

#### genericsensor→get\_hardwareId()

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



**genericSensor→get\_highestValue()**

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

**genericSensor→get\_logFrequency()**

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

**genericSensor→get\_logicalName()**

Returns the logical name of the generic sensor.

**genericSensor→get\_lowestValue()**

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

**genericSensor→get\_module()**

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

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

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

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

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

**genericSensor→get\_reportFrequency()**

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

**genericSensor→get\_resolution()**

Returns the resolution of the measured values.

**genericSensor→get\_signalBias()**

Returns the electric signal bias for zero shift adjustment.

**genericSensor→get\_signalRange()**

Returns the electric signal range used by the sensor.

**genericSensor→get\_signalUnit()**

Returns the measuring unit of the electrical signal used by the sensor.

**genericSensor→get\_signalValue()**

Returns the measured value of the electrical signal used by the sensor.

**genericSensor→get\_unit()**

Returns the measuring unit for the measure.

**genericSensor→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set\_userData.

**genericSensor→get\_valueRange()**

Returns the physical value range measured by the sensor.

**genericSensor→isOnline()**

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

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

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

**genericSensor→load(msValidity)**

Preloads the generic sensor cache with a specified validity duration.

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

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

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

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

**genericSensor→nextGenericSensor()**

Continues the enumeration of generic sensors started using yFirstGenericSensor().

### 3. Reference

**genericsensor**→**registerTimedReportCallback(callback)**

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\_signalBias(newval)**

Changes the electric signal bias for zero shift adjustment.

**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.

**genericsensor**→**zeroAdjust()**

Adjusts the signal bias so that the current signal value is need precisely as zero.

---

**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)=RELAYLO1-123456.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)=RELAYLO1-123456.relay1`)

---

**genericsensor**→**get\_advertisedValue()****YGenericSensor****genericsensor**→**advertisedValue()****genericsensor.get\_advertisedValue()**

---

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()**

**YGenericSensor**

**genericsensor**→**currentRawValue()**

**genericsensor.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`.



---

**genericsensor→get\_currentValue()****YGenericSensor****genericsensor→currentValue()****genericsensor.get\_currentValue()**

---

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 example: `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`.

**genericsensor**→**get\_functionId()**  
**genericsensor**→**functionId()**  
**genericsensor.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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the generic sensor (ex: `RELAYLO1-123456.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()****YGenericSensor****genericsensor**→**logFrequency()****genericsensor.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`.

**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()****YGenericSensor****genericsensor**→**lowestValue()****genericsensor.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`.

**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 on-line.

**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 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.

**genericsensor→get\_reportFrequency()**

**YGenericSensor**

**genericsensor→reportFrequency()**

**genericsensor.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.

---

**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\_signalBias()**

**YGenericSensor**

**genericsensor→signalBias()**

**genericsensor.get\_signalBias()**

---

Returns the electric signal bias for zero shift adjustment.

```
def get_signalBias( )
```

A positive bias means that the signal is over-reporting the measure, while a negative bias means that the signal is underreporting the measure.

**Returns :**

a floating point number corresponding to the electric signal bias for zero shift adjustment

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



---

**genericsensor→get\_signalRange()****YGenericSensor****genericsensor→signalRange()****genericsensor.get\_signalRange()**

---

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()****YGenericSensor****genericsensor**→**signalValue()****genericsensor.get\_signalValue()**

---

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()****YGenericSensor****genericsensor**→**userData()****genericsensor.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.

**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()****YGenericSensor****genericsensor**→**setLogicalName()****genericsensor.set\_logicalName()**

---

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\_signalBias()****YGenericSensor****genericsensor→setSignalBias()****genericsensor.set\_signalBias()**

---

Changes the electric signal bias for zero shift adjustment.

```
def set_signalBias( newval)
```

If your electric signal reads positif when it should be zero, setup a positive signalBias of the same value to fix the zero shift.

**Parameters :**

**newval** a floating point number corresponding to the electric signal bias for zero shift adjustment

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**genericsensor**→**set\_signalRange()**

**YGenericSensor**

**genericsensor**→**setSignalRange()**

**genericsensor.set\_signalRange()**

---

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 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

---

**genericsensor**→**set\_valueRange()**  
**genericsensor**→**setValueRange()**  
**genericsensor.set\_valueRange()**

---

**YGenericSensor**

Changes the physical value range measured by the sensor.

```
def set_valueRange( newval)
```

As a side effect, the range modification may automatically modify the display resolution.

**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.

**genericsensor**→**zeroAdjust()**  
**genericsensor.zeroAdjust()**

**YGenericSensor**

---

Adjusts the signal bias so that the current signal value is need precisely as zero.

```
def zeroAdjust( )
```

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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



## 3.19. 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');
c++	#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

#### yFindGyro(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, in degrees per second, as a floating point number.

#### gyro→get\_currentValue()

Returns the current value of the angular velocity, in degrees per second, as a floating point number.

#### 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()

### 3. Reference

Returns the unique hardware identifier of the gyroscope in the form `SERIAL . FUNCTIONID`.

#### **gyro**→**get\_heading()**

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

#### **gyro**→**get\_highestValue()**

Returns the maximal value observed for the angular velocity since the device was started.

#### **gyro**→**get\_logFrequency()**

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

#### **gyro**→**get\_logicalName()**

Returns the logical name of the gyroscope.

#### **gyro**→**get\_lowestValue()**

Returns the minimal value observed for the angular velocity since the device was started.

#### **gyro**→**get\_module()**

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

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

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

#### **gyro**→**get\_pitch()**

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

#### **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.

#### **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.

#### **gyro**→**get\_quaternionY()**

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.

#### **gyro**→**get\_quaternionZ()**

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.

#### **gyro**→**get\_recordedData(startTime, endTime)**

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

#### **gyro**→**get\_reportFrequency()**

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

#### **gyro**→**get\_resolution()**

Returns the resolution of the measured values.

#### **gyro**→**get\_roll()**

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

#### **gyro**→**get\_unit()**

Returns the measuring unit for the angular velocity.

#### **gyro**→**get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

#### **gyro**→**get\_xValue()**

Returns the angular velocity around the X axis of the device, as a floating point number.

**gyro→get\_yValue()**

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()**  
**yFirstGyro()YGyro.FirstGyro()**

---

**YGyro**

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)=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 gyroscope (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.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, in degrees per second, as a floating point number.

```
def get_currentRawValue() ( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in degrees per second, as a floating point number

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, in degrees per second, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the angular velocity, in degrees per second, as a floating point number

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 example: `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`.

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

---

**YGyro**

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`.

**gyro**→**get\_hardwareId()**

**YGyro**

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

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the gyroscope (ex: `RELAYLO1-123456.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`.

**gyro**→**get\_logicalName()**

**YGyro**

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

---

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`.

**gyro**→**get\_module()**

**YGyro**

**gyro**→**module()****gyro.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

---

**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.

**gyro**→**get\_quaternionY()**

**YGyro**

**gyro**→**quaternionY()****gyro.get\_quaternionY()**

---

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.

**gyro**→**get\_recordedData()****YGyro****gyro**→**recordedData()****gyro.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.

---

**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.

**gyro**→**get\_unit()**

**YGyro**

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

---

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.

**gyro**→**set\_logFrequency()****YGyro****gyro**→**setLogFrequency()****gyro.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.

---

**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.

**gyro**→**set\_lowestValue()**

**YGyro**

**gyro**→**setLowestValue()****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.

**gyro**→**set\_resolution()**

**YGyro**

**gyro**→**setResolution()****gyro.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.

---

**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( 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

## 3.20. 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');
c++	#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.

**hubport**→**get\_module()**

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

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

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

**hubport**→**get\_portState()**

Returns the current state of the Yocto-hub port.

**hubport**→**get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**hubport**→**isOnline()**

Checks if the Yocto-hub port is currently reachable, without raising any error.

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

Checks if the Yocto-hub port is currently reachable, without raising any error (asynchronous version).

**hubport**→**load(msValidity)**

Preloads the Yocto-hub port cache with a specified validity duration.

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

Preloads the Yocto-hub port cache with a specified validity duration (asynchronous version).

**hubport**→**nextHubPort()**

Continues the enumeration of Yocto-hub ports started using `yFirstHubPort()`.

**hubport**→**registerValueCallback(callback)**

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

**hubport**→**set\_enabled(newval)**

Changes the activation of the Yocto-hub port.

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

Changes the logical name of the Yocto-hub port.

**hubport**→**set\_userData(data)**

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

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

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()**  
**yFirstHubPort()YHubPort.FirstHubPort()**

---

**YHubPort**

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)=RELAYLO1-123456.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 Yocto-hub port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)



---

**hubport**→**get\_advertisedValue()****YHubPort****hubport**→**advertisedValue()****hubport.get\_advertisedValue()**

---

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`.

**hubport**→**get\_baudRate()**

**YHubPort**

**hubport**→**baudRate()****hubport.get\_baudRate()**

---

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

**hubport**→**get\_friendlyName()**

**YHubPort**

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

---

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 example: `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()****YHubPort****hubport**→**functionDescriptor()****hubport.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`.

**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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the Yocto-hub port (ex: `RELAYLO1-123456.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 on-line.

**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.

**hubport**→**set\_enabled()**

**YHubPort**

**hubport**→**setEnabled()****hubport.set\_enabled()**

---

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()****YHubPort****hubport**→**setLogicalName()****hubport.set\_logicalName()**

---

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( 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. 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');
c++	#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 uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number.

#### **humidity→get\_currentValue()**

Returns the current value of the humidity, in %RH, as a floating point number.

#### **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`.

**humidity→get\_highestValue()**

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

**humidity→get\_logFrequency()**

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

**humidity→get\_logicalName()**

Returns the logical name of the humidity sensor.

**humidity→get\_lowestValue()**

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

**humidity→get\_module()**

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

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

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

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

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

**humidity→get\_reportFrequency()**

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

**humidity→get\_resolution()**

Returns the resolution of the measured values.

**humidity→get\_unit()**

Returns the measuring unit for the humidity.

**humidity→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set\_userData.

**humidity→isOnline()**

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

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

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

**humidity→load(msValidity)**

Preloads the humidity sensor cache with a specified validity duration.

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

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

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

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

**humidity→nextHumidity()**

Continues the enumeration of humidity sensors started using yFirstHumidity().

**humidity→registerTimedReportCallback(callback)**

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

**humidity→registerValueCallback(callback)**

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

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

Changes the recorded maximal value observed.

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

Changes the datalogger recording frequency for this function.

**humidity→set\_logicalName(newval)**

Changes the logical name of the humidity sensor.

**humidity→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**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() humidity.calibrateFromPoints()

YHumidity

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)=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 humidity sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.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()`**YHumidity****humidity**→`currentRawValue()`**humidity.get\_currentRawValue()

---**

Returns the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number.

```
def get_currentRawValue( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number

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

humidity→get\_currentValue()

YHumidity

humidity→currentValue()humidity.get\_currentValue()

---

Returns the current value of the humidity, in %RH, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the humidity, in %RH, as a floating point number

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

---

**humidity**→**get\_errorMessage()****YHumidity****humidity**→**errorMessage()****humidity.get\_errorMessage()**

---

Returns the error message of the latest error with the humidity 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 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()****YHumidity****humidity→friendlyName()****humidity.get\_friendlyName()**

---

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 example: `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`.

**humidity**→**get\_functionDescriptor()**

**YHumidity**

**humidity**→**functionDescriptor()**

**humidity.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`.

---

**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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the humidity sensor (ex: `RELAYLO1-123456.relay1`)

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

---

**humidity**→**get\_highestValue()****YHumidity****humidity**→**highestValue()****humidity.get\_highestValue()**

---

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

```
def get_highestValue( )
```

**Returns :**

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

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

humidity→get\_logFrequency()

YHumidity

humidity→logFrequency()

humidity.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.

---

**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 since the device was started.

```
def get_lowestValue( )
```

**Returns :**

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

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 on-line.

**Returns :**

an instance of `YModule`

**humidity**→**get\_recordedData()****YHumidity****humidity**→**recordedData()****humidity.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.

---

**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()**  
**humidity.loadCalibrationPoints()**

---

**YHumidity**

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.

```
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.

**humidity**→**set\_logFrequency()**

**YHumidity**

**humidity**→**setLogFrequency()**

**humidity.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.

---

**humidity**→**set\_logicalName()****YHumidity****humidity**→**setLogicalName()****humidity.set\_logicalName()**

---

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()

YHumidity

humidity→setLowestValue()

humidity.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.



---

**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

## 3.22. 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');
c++	#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()****YLed****yFindLed()YLed.FindLed()**

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()****YLed****yFirstLed()YLed.FirstLed()**

---

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` 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)=RELAYLO1-123456.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 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()**

**YLed**

**led**→**blinking()****led.get\_blinking()**

---

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**→`get_errorMessage()`**YLed****led**→`errorMessage()`**led**.`get_errorMessage()`

---

Returns the error message of the latest error with the 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 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 example: `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()**

**YLed**

**led**→**functionDescriptor()**

**led.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`.

---

**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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the led (ex: `RELAYLO1-123456.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 on-line.

**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()**

**YLed**

**led**→**setUserData()****led.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

## 3.23. 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>
nodejs	var yoctolib = require('yoctolib'); var YLightSensor = yoctolib.YLightSensor;
php	require_once('yocto_lightsensor.php');
cpp	#include "yocto_lightsensor.h"
m	#import "yocto_lightsensor.h"
pas	uses yocto_lightsensor;
vb	yocto_lightsensor.vb
cs	yocto_lightsensor.cs
java	import com.yoctopuce.YoctoAPI.YLightSensor;
py	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 uncalibrated, unrounded raw value returned by the sensor, in lux, as a floating point number.

#### lightsensor→get\_currentValue()

Returns the current value of the ambient light, in lux, as a floating point number.

#### 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.

**lightsensor**→**get\_hardwareId()**

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

**lightsensor**→**get\_highestValue()**

Returns the maximal value observed for the ambient light since the device was started.

**lightsensor**→**get\_logFrequency()**

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

**lightsensor**→**get\_logicalName()**

Returns the logical name of the light sensor.

**lightsensor**→**get\_lowestValue()**

Returns the minimal value observed for the ambient light since the device was started.

**lightsensor**→**get\_measureType()**

Returns the type of light measure.

**lightsensor**→**get\_module()**

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

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

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

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

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

**lightsensor**→**get\_reportFrequency()**

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

**lightsensor**→**get\_resolution()**

Returns the resolution of the measured values.

**lightsensor**→**get\_unit()**

Returns the measuring unit for the ambient light.

**lightsensor**→**get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**lightsensor**→**isOnline()**

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

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

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

**lightsensor**→**load(msValidity)**

Preloads the light sensor cache with a specified validity duration.

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

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

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

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

**lightsensor**→**nextLightSensor()**

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

**lightsensor**→**registerTimedReportCallback(callback)**

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

**lightsensor**→**registerValueCallback(callback)**

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

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



Changes the recorded maximal value observed.

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

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.

**lightsensor**→**set\_measureType(newval)**

Modify the light sensor type used in the device.

**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()  
yFirstLightSensor()YLightSensor.FirstLightSensor()**

---

**YLightSensor**

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(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 light sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

---

**lightsensor**→**get\_advertisedValue()****YLightSensor****lightsensor**→**advertisedValue()****lightsensor.get\_advertisedValue()**

---

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()**

**YLightSensor**

**lightsensor**→**currentRawValue()**

**lightsensor.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in lux, as a floating point number.

```
def get_currentRawValue( )
```

**Returns :**

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

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



---

**lightsensor**→**get\_currentValue()****YLightSensor****lightsensor**→**currentValue()****lightsensor.get\_currentValue()**

---

Returns the current value of the ambient light, in lux, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the ambient light, in lux, as a floating point number

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

**lightsensor→get\_errorMessage()**

**YLightSensor**

**lightsensor→errorMessage()**

**lightsensor.get\_errorMessage()**

---

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()**

**YLightSensor**

**lightsensor→friendlyName()**

**lightsensor.get\_friendlyName()**

---

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 example: `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()**  
**lightsensor→functionDescriptor()**  
**lightsensor.get\_functionDescriptor()**

---

**YLightSensor**

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\_hardwareId()****YLightSensor****lightsensor**→**hardwareId()****lightsensor.get\_hardwareId()**

---

Returns the unique hardware identifier of the light 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 light sensor (for example `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the light sensor (ex: `RELAYLO1-123456.relay1`)

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

lightsensor→get\_highestValue()

YLightSensor

lightsensor→highestValue()

lightsensor.get\_highestValue()

---

Returns the maximal value observed for the ambient light since the device was started.

```
def get_highestValue( )
```

**Returns :**

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

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



---

**lightsensor→get\_logFrequency()****YLightSensor****lightsensor→logFrequency()****lightsensor.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.

**lightsensor**→**get\_logicalName()**

**YLightSensor**

**lightsensor**→**logicalName()**

**lightsensor.get\_logicalName()**

---

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()****YLightSensor****lightsensor**→**lowestValue()****lightsensor.get\_lowestValue()**

---

Returns the minimal value observed for the ambient light since the device was started.

```
def get_lowestValue( )
```

**Returns :**

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

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

**lightsensor**→**get\_measureType()**  
**lightsensor**→**measureType()**  
**lightsensor.get\_measureType()**

---

**YLightSensor**

Returns the type of light measure.

```
def get_measureType( )
```

**Returns :**

a value among `Y_MEASURETYPE_HUMAN_EYE`, `Y_MEASURETYPE_WIDE_SPECTRUM`, `Y_MEASURETYPE_INFRARED`, `Y_MEASURETYPE_HIGH_RATE` and `Y_MEASURETYPE_HIGH_ENERGY` corresponding to the type of light measure

On failure, throws an exception or returns `Y_MEASURETYPE_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 on-line.

**Returns :**

an instance of `YModule`

**lightsensor**→**get\_recordedData()****YLightSensor****lightsensor**→**recordedData()****lightsensor.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.

---

**lightsensor**→**get\_reportFrequency()****YLightSensor****lightsensor**→**reportFrequency()****lightsensor.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`.

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()****YLightSensor****lightsensor**→**setHighestValue()****lightsensor.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.

**lightsensor**→**set\_logFrequency()**

**YLightSensor**

**lightsensor**→**setLogFrequency()**

**lightsensor.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.

---

**lightsensor**→**set\_logicalName()****YLightSensor****lightsensor**→**setLogicalName()****lightsensor.set\_logicalName()**

---

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()

YLightSensor

lightsensor→setLowestValue()

lightsensor.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.

---

**lightsensor**→**set\_measureType()****YLightSensor****lightsensor**→**setMeasureType()****lightsensor.set\_measureType()**

---

Modify the light sensor type used in the device.

```
def set_measureType( newval)
```

The measure can either approximate the response of the human eye, focus on a specific light spectrum, depending on the capabilities of the light-sensitive cell. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a value among `Y_MEASURETYPE_HUMAN_EYE`, `Y_MEASURETYPE_WIDE_SPECTRUM`, `Y_MEASURETYPE_INFRARED`, `Y_MEASURETYPE_HIGH_RATE` and `Y_MEASURETYPE_HIGH_ENERGY`

**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



## 3.24. 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'); var YMagnetometer = yoctolib.YMagnetometer;
php	require_once('yocto_magnetometer.php');
c++	#include "yocto_magnetometer.h"
m	#import "yocto_magnetometer.h"
pas	uses yocto_magnetometer;
vb	yocto_magnetometer.vb
cs	yocto_magnetometer.cs
java	import com.yoctopuce.YoctoAPI.YMagnetometer;
py	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, in mT, as a floating point number.

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

Returns the current value of the magnetic field, in mT, as a floating point number.

#### **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`.

**magnetometer**→**get\_highestValue()**

Returns the maximal value observed for the magnetic field since the device was started.

**magnetometer**→**get\_logFrequency()**

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

**magnetometer**→**get\_logicalName()**

Returns the logical name of the magnetometer.

**magnetometer**→**get\_lowestValue()**

Returns the minimal value observed for the magnetic field since the device was started.

**magnetometer**→**get\_module()**

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

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

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

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

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

**magnetometer**→**get\_reportFrequency()**

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

**magnetometer**→**get\_resolution()**

Returns the resolution of the measured values.

**magnetometer**→**get\_unit()**

Returns the measuring unit for the magnetic field.

**magnetometer**→**get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**magnetometer**→**get\_xValue()**

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

**magnetometer**→**get\_yValue()**

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

**magnetometer**→**get\_zValue()**

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

**magnetometer**→**isOnline()**

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

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

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

**magnetometer**→**load(msValidity)**

Preloads the magnetometer cache with a specified validity duration.

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

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

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

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

**magnetometer**→**nextMagnetometer()**

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

**magnetometer**→**registerTimedReportCallback(callback)**

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

**magnetometer**→**registerValueCallback(callback)**

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.

**YMagnetometer.FindMagnetometer()  
yFindMagnetometer()  
YMagnetometer.FindMagnetometer()****YMagnetometer**

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 `YMagnetometer.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 `YMagnetometer` object allowing you to drive the magnetometer.

---

**YMagnetometer.FirstMagnetometer()**  
**yFirstMagnetometer()**  
**YMagnetometer.FirstMagnetometer()**

---

**YMagnetometer**

Starts the enumeration of magnetometers currently accessible.

```
def FirstMagnetometer( )
```

Use the method `YMagnetometer.nextMagnetometer( )` to iterate on next magnetometers.

**Returns :**

a pointer to a `YMagnetometer` 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)=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 magnetometer (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

magnetometer→get\_advertisedValue()

YMagnetometer

magnetometer→advertisedValue()

magnetometer.get\_advertisedValue()

---

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()****YMagnetometer****magnetometer**→**currentRawValue()****magnetometer.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in mT, as a floating point number.

```
def get_currentRawValue( )
```

**Returns :**

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

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

magnetometer→get\_currentValue()

YMagnetometer

magnetometer→currentValue()

magnetometer.get\_currentValue()

---

Returns the current value of the magnetic field, in mT, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the magnetic field, in mT, as a floating point number

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

---

**magnetometer**→**get\_errorMessage()****YMagnetometer****magnetometer**→**errorMessage()****magnetometer.get\_errorMessage()**

---

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()****YMagnetometer****magnetometer**→**friendlyName()****magnetometer.get\_friendlyName()**

---

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 example: `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()**

**YMagnetometer**

**magnetometer**→**functionDescriptor()**

**magnetometer.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`.

---

**magnetometer**→**get\_functionId()****YMagnetometer****magnetometer**→**functionId()****magnetometer.get\_functionId()**

---

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()`

**YMagnetometer**

`magnetometer`→`hardwareId()`

`magnetometer.get_hardwareId()`

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the magnetometer (ex: `RELAYLO1-123456.relay1`)

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



---

**magnetometer**→**get\_highestValue()****YMagnetometer****magnetometer**→**highestValue()****magnetometer.get\_highestValue()**

---

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()

YMagnetometer

magnetometer→logFrequency()

magnetometer.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.

---

**magnetometer**→**get\_logicalName()****YMagnetometer****magnetometer**→**logicalName()****magnetometer.get\_logicalName()**

---

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()

YMagnetometer

magnetometer→lowestValue()

magnetometer.get\_lowestValue()

---

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 on-line.

**Returns :**

an instance of `YModule`

**magnetometer**→**get\_recordedData()**

**YMagnetometer**

**magnetometer**→**recordedData()**

**magnetometer.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.

---

**magnetometer**→**get\_reportFrequency()****YMagnetometer****magnetometer**→**reportFrequency()****magnetometer.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`.

**magnetometer**→**get\_resolution()**

**YMagnetometer**

**magnetometer**→**resolution()**

**magnetometer.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`.



---

**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()**

**YMagnetometer**

**magnetometer**→**userData()**

**magnetometer.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.

---

**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()**

---

**YMagnetometer**

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

```
def nextMagnetometer()
```

**Returns :**

a pointer to a `YMagnetometer` 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()

YMagnetometer

magnetometer→setHighestValue()

magnetometer.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.

---

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

---

**YMagnetometer**

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()**

**YMagnetometer**

**magnetometer→setLogicalName()**

**magnetometer.set\_logicalName()**

---

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()****YMagnetometer****magnetometer**→**setLowestValue()****magnetometer.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.

**magnetometer→set\_reportFrequency()**

**YMagnetometer**

**magnetometer→setReportFrequency()**

**magnetometer.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.



**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()**

**YMagnetometer**

**magnetometer→setUserData()**

**magnetometer.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

## 3.25. 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>
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 *

### 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()**

**YMeasure**

**measure**→**averageValue()**

**measure.get\_averageValue()**

---

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 than 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→get_startTimeUTC()`

**YMeasure**

`measure→startTimeUTC()`

`measure.get_startTimeUTC()`

---

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

```
def get_startTimeUTC( )
```

When the recording rate is higher than 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.



## 3.26. 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>
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

#### 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→checkFirmware(path, onlynew)

Test if the byn file is valid for this module.

#### 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\_allSettings()

Returns all the setting of 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.
<b>module</b> → <b>get_hardwareId()</b>	Returns the unique hardware identifier of the module.
<b>module</b> → <b>get_icon2d()</b>	Returns the icon of the module.
<b>module</b> → <b>get_lastLogs()</b>	Returns a string with last logs of the module.
<b>module</b> → <b>get_logicalName()</b>	Returns the logical name of the module.
<b>module</b> → <b>get_luminosity()</b>	Returns the luminosity of the module informative leds (from 0 to 100).
<b>module</b> → <b>get_persistentSettings()</b>	Returns the current state of persistent module settings.
<b>module</b> → <b>get_productId()</b>	Returns the USB device identifier of the module.
<b>module</b> → <b>get_productName()</b>	Returns the commercial name of the module, as set by the factory.
<b>module</b> → <b>get_productRelease()</b>	Returns the hardware release version of the module.
<b>module</b> → <b>get_rebootCountdown()</b>	Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.
<b>module</b> → <b>get_serialNumber()</b>	Returns the serial number of the module, as set by the factory.
<b>module</b> → <b>get_upTime()</b>	Returns the number of milliseconds spent since the module was powered on.
<b>module</b> → <b>get_usbCurrent()</b>	Returns the current consumed by the module on the USB bus, in milli-amps.
<b>module</b> → <b>get_userData()</b>	Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
<b>module</b> → <b>get_userVar()</b>	Returns the value previously stored in this attribute.
<b>module</b> → <b>isOnline()</b>	Checks if the module is currently reachable, without raising any error.
<b>module</b> → <b>isOnline_async(callback, context)</b>	Checks if the module is currently reachable, without raising any error.
<b>module</b> → <b>load(msValidity)</b>	Preloads the module cache with a specified validity duration.
<b>module</b> → <b>load_async(msValidity, callback, context)</b>	Preloads the module cache with a specified validity duration (asynchronous version).
<b>module</b> → <b>nextModule()</b>	Continues the module enumeration started using <code>yFirstModule()</code> .
<b>module</b> → <b>reboot(secBeforeReboot)</b>	Schedules a simple module reboot after the given number of seconds.
<b>module</b> → <b>registerLogCallback(callback)</b>	Registers a device log callback function.

**module→revertFromFlash()**

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

**module→saveToFlash()**

Saves current settings in the nonvolatile memory of the module.

**module→set\_allSettings(settings)**

Restore all the setting 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\_userData(data)**

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

**module→set\_userVar(newval)**

Returns the value previously stored in this attribute.

**module→triggerFirmwareUpdate(secBeforeReboot)**

Schedules a module reboot into special firmware update mode.

**module→updateFirmware(path)**

Prepare a firmware upgrade of the module.

**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()****YModule****yFindModule()YModule.FindModule()**

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()**  
**yFirstModule()YModule.FirstModule()**

---

**YModule**

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**→**checkFirmware()****module.checkFirmware()**

---

**YModule**

Test if the byn file is valid for this module.

```
def checkFirmware( path, onlynew)
```

This method is useful to test if the module need to be updated. It's possible to pass an directory instead of a file. In this case this method return the path of the most recent appropriate byn file. If the parameter `onlynew` is true the function will discard firmware that are older or equal to the installed firmware.

**Parameters :**

- path** the path of a byn file or a directory that contain byn files
- onlynew** return only files that are strictly newer

**Returns :**

: the path of the byn file to use or a empty string if no byn files match the requirement

On failure, throws an exception or returns a string that start with "error:".

---

**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 `YAPI_INVALID_STRING`.



---

**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\_allSettings()****YModule****module**→**allSettings()****module.get\_allSettings()**

---

Returns all the setting of the module.

```
def get_allSettings( )
```

Useful to backup all the logical name and calibrations parameters of a connected module.

**Returns :**

a binary buffer with all settings.

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

**module**→**get\_beacon()**

**YModule**

**module**→**beacon()****module.get\_beacon()**

---

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**→**get\_errorMessage()****YModule****module**→**errorMessage()****module.get\_errorMessage()**

---

Returns the error message of the latest error with this module object.

```
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 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()****YModule****module**→**firmwareRelease()****module.get\_firmwareRelease()**

---

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()**

**YModule**

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

---

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

**module**→**get\_lastLogs()**

**YModule**

**module**→**lastLogs()****module.get\_lastLogs()**

---

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. On failure, throws an exception or returns YAPI\_INVALID\_STRING.

---

**module**→**get\_logicalName()****YModule****module**→**logicalName()****module.get\_logicalName()**

---

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()****YModule****module**→**persistentSettings()****module.get\_persistentSettings()**

---

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`.



---

**module**→**get\_productName()****YModule****module**→**productName()****module.get\_productName()**

---

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()**

**YModule**

**module**→**productRelease()**

**module.get\_productRelease()**

---

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()****YModule****module**→**rebootCountdown()****module.get\_rebootCountdown()**

---

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`.

---

**module**→**get\_upTime()****YModule****module**→**upTime()****module.get\_upTime()**

---

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\_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()****YModule****module**→**userData()****module.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.

**module**→**get\_userVar()**

**YModule**

**module**→**userVar()****module.get\_userVar()**

---

Returns the value previously stored in this attribute.

```
def get_userVar( )
```

On startup and after a device reboot, the value is always reset to zero.

**Returns :**

an integer corresponding to the value previously stored in this attribute

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



---

**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**

Registers a device log callback function.

```
def registerLogCallback( callback )
```

This callback will be called each time that a module sends a new log message. Mostly useful to debug a Yoctopuce module.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the module object that emitted the log message, and the character string containing the log.

**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\_allSettings()**

**YModule**

**module**→**setAllSettings()****module.set\_allSettings()**

---

Restore all the setting of the module.

```
def set_allSettings( settings)
```

Useful to restore all the logical name and calibrations parameters of a module from a backup.

**Parameters :**

**settings** a binary buffer with all settings.

**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.

**module**→**set\_logicalName()**

**YModule**

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

---

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\_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**→**set\_userVar()****YModule****module**→**setUserVar()****module.set\_userVar()**

---

Returns the value previously stored in this attribute.

```
def set_userVar( newval)
```

On startup and after a device reboot, the value is always reset to zero.

**Parameters :**

**newval** an integer

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**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.

---

**module**→**updateFirmware()****module.updateFirmware()****YModule**

---

Prepare a firmware upgrade of the module.

```
def updateFirmware( path)
```

This method return a object `YFirmwareUpdate` which will handle the firmware upgrade process.

**Parameters :**

**path** the path of the byn file to use.

**Returns :**

: A object `YFirmwareUpdate`.

## 3.27. Motor function interface

Yoctopuce application programming interface allows you to drive the power sent to the motor to make it turn both ways, but also to drive accelerations and decelerations. The motor will then accelerate automatically: you will not have to monitor it. The API also allows to slow down the motor by shortening its terminals: the motor will then act as an electromagnetic brake.

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

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

### Global functions

#### **yFindMotor(func)**

Retrieves a motor for a given identifier.

#### **yFirstMotor()**

Starts the enumeration of motors currently accessible.

### YMotor methods

#### **motor→brakingForceMove(targetPower, delay)**

Changes progressively the braking force applied to the motor for a specific duration.

#### **motor→describe()**

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

#### **motor→drivingForceMove(targetPower, delay)**

Changes progressively the power sent to the moteur for a specific duration.

#### **motor→get\_advertisedValue()**

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

#### **motor→get\_brakingForce()**

Returns the braking force applied to the motor, as a percentage.

#### **motor→get\_cutOffVoltage()**

Returns the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

#### **motor→get\_drivingForce()**

Returns the power sent to the motor, as a percentage between -100% and +100%.

#### **motor→get\_errorMessage()**

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

#### **motor→get\_errorType()**

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

#### **motor→get\_failSafeTimeout()**

Returns the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.



**motor**→**get\_frequency()**

Returns the PWM frequency used to control the motor.

**motor**→**get\_friendlyName()**

Returns a global identifier of the motor in the format `MODULE_NAME . FUNCTION_NAME`.

**motor**→**get\_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**motor**→**get\_functionId()**

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

**motor**→**get\_hardwareId()**

Returns the unique hardware identifier of the motor in the form `SERIAL . FUNCTIONID`.

**motor**→**get\_logicalName()**

Returns the logical name of the motor.

**motor**→**get\_module()**

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

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

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

**motor**→**get\_motorStatus()**

Return the controller state.

**motor**→**get\_overCurrentLimit()**

Returns the current threshold (in mA) above which the controller automatically switches to error state.

**motor**→**get\_starterTime()**

Returns the duration (in ms) during which the motor is driven at low frequency to help it start up.

**motor**→**get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**motor**→**isOnline()**

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

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

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

**motor**→**keepALive()**

Rearms the controller failsafe timer.

**motor**→**load(msValidity)**

Preloads the motor cache with a specified validity duration.

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

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

**motor**→**nextMotor()**

Continues the enumeration of motors started using `yFirstMotor()`.

**motor**→**registerValueCallback(callback)**

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

**motor**→**resetStatus()**

Reset the controller state to IDLE.

**motor**→**set\_brakingForce(newval)**

Changes immediately the braking force applied to the motor (in percents).

**motor**→**set\_cutOffVoltage(newval)**

Changes the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

### 3. Reference

---

**motor**→**set\_drivingForce(newval)**

Changes immediately the power sent to the motor.

**motor**→**set\_failSafeTimeout(newval)**

Changes the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

**motor**→**set\_frequency(newval)**

Changes the PWM frequency used to control the motor.

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

Changes the logical name of the motor.

**motor**→**set\_overCurrentLimit(newval)**

Changes the current threshold (in mA) above which the controller automatically switches to error state.

**motor**→**set\_starterTime(newval)**

Changes the duration (in ms) during which the motor is driven at low frequency to help it start up.

**motor**→**set\_userData(data)**

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

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

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

## YMotor.FindMotor() yFindMotor() YMotor.FindMotor()

YMotor

Retrieves a motor for a given identifier.

```
def FindMotor( 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 motor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YMotor.isOnline()` to test if the motor is indeed online at a given time. In case of ambiguity when looking for a motor 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 motor

**Returns :**

a `YMotor` object allowing you to drive the motor.

## **YMotor.FirstMotor() yFirstMotor() YMotor.FirstMotor()**

---

**YMotor**

Starts the enumeration of motors currently accessible.

```
def FirstMotor( )
```

Use the method `YMotor.nextMotor()` to iterate on next motors.

**Returns :**

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

---

**motor**→**brakingForceMove()**  
**motor.brakingForceMove()**

---

**YMotor**

Changes progressively the braking force applied to the motor for a specific duration.

```
def brakingForceMove( targetPower, delay)
```

**Parameters :**

**targetPower** desired braking force, in percents

**delay** duration (in ms) of the transition

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**motor**→**describe()****motor.describe()****YMotor**

Returns a short text that describes unambiguously the instance of the motor 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(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 motor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

---

**motor**→**drivingForceMove()**  
**motor.drivingForceMove()**

---

**YMotor**

Changes progressively the power sent to the moteur for a specific duration.

```
def drivingForceMove( targetPower, delay)
```

**Parameters :**

**targetPower** desired motor power, in percents (between -100% and +100%)

**delay** duration (in ms) of the transition

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**motor**→**get\_advertisedValue()**

**YMotor**

**motor**→**advertisedValue()**

**motor.get\_advertisedValue()**

---

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

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the motor (no more than 6 characters).

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



---

**motor**→**get\_brakingForce()****YMotor****motor**→**brakingForce()****motor.get\_brakingForce()**

---

Returns the braking force applied to the motor, as a percentage.

```
def get_brakingForce( )
```

The value 0 corresponds to no braking (free wheel).

**Returns :**

a floating point number corresponding to the braking force applied to the motor, as a percentage

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

**motor**→**get\_cutOffVoltage()**

**YMotor**

**motor**→**cutOffVoltage()****motor.get\_cutOffVoltage()**

---

Returns the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

```
def get_cutOffVoltage( )
```

This setting prevents damage to a battery that can occur when drawing current from an "empty" battery.

**Returns :**

a floating point number corresponding to the threshold voltage under which the controller automatically switches to error state and prevents further current draw

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

---

**motor**→**get\_drivingForce()****YMotor****motor**→**drivingForce()****motor.get\_drivingForce()**

---

Returns the power sent to the motor, as a percentage between -100% and +100%.

```
def get_drivingForce( )
```

**Returns :**

a floating point number corresponding to the power sent to the motor, as a percentage between -100% and +100%

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

**motor**→**get\_errorMessage()**

**YMotor**

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

---

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

```
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 motor object

---

**motor**→**get\_errorType()****YMotor****motor**→**errorType()****motor.get\_errorType()**

---

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

```
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 motor object

**motor**→**get\_failSafeTimeout()**

**YMotor**

**motor**→**failSafeTimeout()****motor.get\_failSafeTimeout()**

---

Returns the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

```
def get_failSafeTimeout( )
```

When this delay has elapsed, the controller automatically stops the motor and switches to FAILSAFE error. Failsafe security is disabled when the value is zero.

**Returns :**

an integer corresponding to the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process

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

---

**motor**→**get\_frequency()****YMotor****motor**→**frequency()****motor.get\_frequency()**

---

Returns the PWM frequency used to control the motor.

```
def get_frequency( )
```

**Returns :**

a floating point number corresponding to the PWM frequency used to control the motor

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

**motor**→**get\_friendlyName()**

**YMotor**

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

---

Returns a global identifier of the motor in the format `MODULE_NAME.FUNCTION_NAME`.

```
def get_friendlyName( )
```

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

**Returns :**

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

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



---

**motor**→**get\_functionDescriptor()****YMotor****motor**→**functionDescriptor()****motor.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`.

**motor**→**get\_functionId()**

**YMotor**

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

---

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

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the motor (ex: `relay1`)

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

---

**motor**→`get_hardwareId()`**YMotor****motor**→`hardwareId()`**motor.get\_hardwareId()**

---

Returns the unique hardware identifier of the motor 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 motor (for example `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the motor (ex: `RELAYLO1-123456.relay1`)

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

**motor**→**get\_logicalName()**

**YMotor**

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

---

Returns the logical name of the motor.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the motor.

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

---

**motor**→**get\_module()****YMotor****motor**→**module()****motor.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

**motor**→**get\_motorStatus()****YMotor****motor**→**motorStatus()****motor.get\_motorStatus()**

Return the controller state.

```
def get_motorStatus( )
```

Possible states are: IDLE when the motor is stopped/in free wheel, ready to start; FORWD when the controller is driving the motor forward; BACKWD when the controller is driving the motor backward; BRAKE when the controller is braking; LOVOLT when the controller has detected a low voltage condition; HICURR when the controller has detected an overcurrent condition; HIHEAT when the controller has detected an overheat condition; FAILSF when the controller switched on the failsafe security.

When an error condition occurred (LOVOLT, HICURR, HIHEAT, FAILSF), the controller status must be explicitly reset using the `resetStatus` function.

**Returns :**

a value among `Y_MOTORSTATUS_IDLE`, `Y_MOTORSTATUS_BRAKE`, `Y_MOTORSTATUS_FORWD`, `Y_MOTORSTATUS_BACKWD`, `Y_MOTORSTATUS_LOVOLT`, `Y_MOTORSTATUS_HICURR`, `Y_MOTORSTATUS_HIHEAT` and `Y_MOTORSTATUS_FAILSF`

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

---

**motor**→**get\_overCurrentLimit()****YMotor****motor**→**overCurrentLimit()****motor.get\_overCurrentLimit()**

---

Returns the current threshold (in mA) above which the controller automatically switches to error state.

```
def get_overCurrentLimit( )
```

A zero value means that there is no limit.

**Returns :**

an integer corresponding to the current threshold (in mA) above which the controller automatically switches to error state

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

**motor**→**get\_starterTime()**

**YMotor**

**motor**→**starterTime()****motor.get\_starterTime()**

---

Returns the duration (in ms) during which the motor is driven at low frequency to help it start up.

```
def get_starterTime( )
```

**Returns :**

an integer corresponding to the duration (in ms) during which the motor is driven at low frequency to help it start up

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



---

**motor**→**get\_userData()****YMotor****motor**→**userData()****motor.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.

**motor**→**isOnline()****motor.isOnline()**

**YMotor**

---

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

```
def isOnline( )
```

If there is a cached value for the motor 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 motor.

**Returns :**

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

---

**motor**→**keepALive()****motor.keepALive()****YMotor**

---

Rearms the controller failsafe timer.

```
def keepALive( )
```

When the motor is running and the failsafe feature is active, this function should be called periodically to prove that the control process is running properly. Otherwise, the motor is automatically stopped after the specified timeout. Calling a motor `set` function implicitly rearms the failsafe timer.

**motor**→**load()****motor.load()****YMotor**

Preloads the motor 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.

---

**motor**→**nextMotor()****motor.nextMotor()****YMotor**

---

Continues the enumeration of motors started using `yFirstMotor()`.

```
def nextMotor( )
```

**Returns :**

a pointer to a `YMotor` object, corresponding to a motor currently online, or a `null` pointer if there are no more motors to enumerate.

**motor**→**registerValueCallback()**  
**motor.registerValueCallback()****YMotor**

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.

---

**motor**→**resetStatus()****motor.resetStatus()**

---

**YMotor**

Reset the controller state to IDLE.

```
def resetStatus( )
```

This function must be invoked explicitly after any error condition is signaled.

**motor**→**set\_brakingForce()**

**YMotor**

**motor**→**setBrakingForce()****motor.set\_brakingForce()**

---

Changes immediately the braking force applied to the motor (in percents).

```
def set_brakingForce( newval)
```

The value 0 corresponds to no braking (free wheel). When the braking force is changed, the driving power is set to zero. The value is a percentage.

**Parameters :**

**newval** a floating point number corresponding to immediately the braking force applied to the motor (in percents)

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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



---

**motor**→**set\_cutOffVoltage()****YMotor****motor**→**setCutOffVoltage()****motor.set\_cutOffVoltage()**

---

Changes the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

```
def set_cutOffVoltage( newval)
```

This setting prevent damage to a battery that can occur when drawing current from an "empty" battery. Note that whatever the cutoff threshold, the controller switches to undervoltage error state if the power supply goes under 3V, even for a very brief time.

**Parameters :**

**newval** a floating point number corresponding to the threshold voltage under which the controller automatically switches to error state and prevents further current draw

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**motor**→**set\_drivingForce()**

**YMotor**

**motor**→**setDrivingForce()****motor.set\_drivingForce()**

---

Changes immediately the power sent to the motor.

```
def set_drivingForce( newval)
```

The value is a percentage between -100% to 100%. If you want go easy on your mechanics and avoid excessive current consumption, try to avoid brutal power changes. For example, immediate transition from forward full power to reverse full power is a very bad idea. Each time the driving power is modified, the braking power is set to zero.

**Parameters :**

**newval** a floating point number corresponding to immediately the power sent to the motor

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**motor**→**set\_failSafeTimeout()**  
**motor**→**setFailSafeTimeout()**  
**motor.set\_failSafeTimeout()**

---

**YMotor**

Changes the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

```
def set_failSafeTimeout( newval)
```

When this delay has elapsed, the controller automatically stops the motor and switches to FAILSAFE error. Failsafe security is disabled when the value is zero.

**Parameters :**

**newval** an integer corresponding to the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**motor**→**set\_frequency()**

**YMotor**

**motor**→**setFrequency()****motor.set\_frequency()**

---

Changes the PWM frequency used to control the motor.

```
def set_frequency( newval)
```

Low frequency is usually more efficient and may help the motor to start, but an audible noise might be generated. A higher frequency reduces the noise, but more energy is converted into heat.

**Parameters :**

**newval** a floating point number corresponding to the PWM frequency used to control the motor

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**motor**→**set\_logicalName()****YMotor****motor**→**setLogicalName()****motor.set\_logicalName()**

---

Changes the logical name of the motor.

```
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 motor.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**motor**→**set\_overCurrentLimit()**

**YMotor**

**motor**→**setOverCurrentLimit()**

**motor.set\_overCurrentLimit()**

---

Changes the current threshold (in mA) above which the controller automatically switches to error state.

```
def set_overCurrentLimit( newval)
```

A zero value means that there is no limit. Note that whatever the current limit is, the controller switches to OVERCURRENT status if the current goes above 32A, even for a very brief time.

**Parameters :**

**newval** an integer corresponding to the current threshold (in mA) above which the controller automatically switches to error state

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**motor**→**set\_starterTime()****YMotor****motor**→**setStarterTime()****motor.set\_starterTime()**

---

Changes the duration (in ms) during which the motor is driven at low frequency to help it start up.

```
def set_starterTime( newval)
```

**Parameters :**

**newval** an integer corresponding to the duration (in ms) during which the motor is driven at low frequency to help it start up

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**motor**→**set\_userData()**

**YMotor**

**motor**→**setUserData()****motor.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



## 3.28. 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).

### 3. Reference

#### **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).

**network→load(msValidity)**

Preloads the network interface cache with a specified validity duration.

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

Preloads the network interface cache with a specified validity duration (asynchronous version).

**network→nextNetwork()**

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

**network→ping(host)**

Pings `str_host` to test the network connectivity.

**network→registerValueCallback(callback)**

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

**network→set\_adminPassword(newval)**

Changes the password for the "admin" user.

**network→set\_callbackCredentials(newval)**

Changes the credentials required to connect to the callback address.

**network→set\_callbackEncoding(newval)**

Changes the encoding standard to use for representing notification values.

**network→set\_callbackMaxDelay(newval)**

Changes the maximum waiting time between two callback notifications, in seconds.

**network→set\_callbackMethod(newval)**

Changes the HTTP method used to notify callbacks for significant state changes.

**network→set\_callbackMinDelay(newval)**

Changes the minimum waiting time between two callback notifications, in seconds.

**network→set\_callbackUrl(newval)**

Changes the callback URL to notify significant state changes.

**network→set\_discoverable(newval)**

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

**network→set\_logicalName(newval)**

Changes the logical name of the network interface.

**network→set\_primaryDNS(newval)**

Changes the IP address of the primary name server to be used by the module.

**network→set\_secondaryDNS(newval)**

Changes the IP address of the secondary name server to be used by the module.

**network→set\_userData(data)**

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

**network→set\_userPassword(newval)**

Changes the password for the "user" user.

**network→set\_wwwWatchdogDelay(newval)**

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

**network→useDHCP(fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)**

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

**network→useStaticIP(ipAddress, subnetMaskLen, router)**

Changes the configuration of the network interface to use a static IP address.

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

### 3. Reference

---

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 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)=RELAYLO1-123456.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)=RELAYLO1-123456.relay1`)



---

**network**→**get\_adminPassword()****YNetwork****network**→**adminPassword()****network.get\_adminPassword()**

---

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()**

**YNetwork**

**network**→**advertisedValue()**

**network.get\_advertisedValue()**

---

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()****YNetwork****network**→**callbackCredentials()****network.get\_callbackCredentials()**

---

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()**

**YNetwork**

**network**→**callbackEncoding()**

**network.get\_callbackEncoding()**

---

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()****YNetwork****network**→**callbackMaxDelay()****network.get\_callbackMaxDelay()**

---

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()**

**YNetwork**

**network**→**callbackMethod()**

**network.get\_callbackMethod()**

---

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()****YNetwork****network**→**callbackMinDelay()****network.get\_callbackMinDelay()**

---

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()**

**YNetwork**

**network**→**errorMessage()**

**network.get\_errorMessage()**

---

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 example: `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()****YNetwork****network**→**functionDescriptor()****network.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`.

**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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the network interface (ex: `RELAYLO1-123456.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`.

**network**→**get\_macAddress()**

**YNetwork**

**network**→**macAddress()****network.get\_macAddress()**

---

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()****YNetwork****network**→**module()****network.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`

**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()**

**YNetwork**

**network→secondaryDNS()**

**network.get\_secondaryDNS()**

---

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()**YNetwork****network**→userPassword()**network.get\_userPassword()**

---

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()**

**YNetwork**

**network**→**wwwWatchdogDelay()**

**network.get\_wwwWatchdogDelay()**

---

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

---

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()****YNetwork****network**→**setCallbackCredentials()****network.set\_callbackCredentials()**

---

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()****YNetwork****network**→**setCallbackMaxDelay()****network.set\_callbackMaxDelay()**

---

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()****YNetwork****network**→**setCallbackMethod()****network.set\_callbackMethod()**

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()****YNetwork****network**→**setCallbackMinDelay()****network.set\_callbackMinDelay()**

---

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()****YNetwork****network**→**setDiscoverable()****network.set\_discoverable()**

---

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()****YNetwork****network**→**setLogicalName()****network.set\_logicalName()**

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()****YNetwork****network**→**setSecondaryDNS()****network.set\_secondaryDNS()**

---

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 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

**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()****YNetwork****network**→**setWwwWatchdogDelay()****network.set\_wwwWatchdogDelay()**

---

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 when 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 when the call succeeds.

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

## 3.29. 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>
nodejs	var yoctolib = require('yoctolib'); var YOsControl = yoctolib.YOsControl;
php	require_once('yocto_oscontrol.php');
c++	#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)

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(data)`**

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()****YOsControl****yFindOsControl()YOsControl.FindOsControl()**

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()**  
**yFirstOsControl()YOsControl.FirstOsControl()**

---

**YOsControl**

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()****oscontrol.describe()****YOsControl**

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` 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)=RELAYLO1-123456.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)=RELAYLO1-123456.relay1`)

---

**oscontrol**→**get\_advertisedValue()****YOsControl****oscontrol**→**advertisedValue()****oscontrol.get\_advertisedValue()**

---

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()**

**YOsControl**

**oscontrol**→**friendlyName()**

**oscontrol.get\_friendlyName()**

---

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 example: `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`.

---

**oscontrol**→**get\_hardwareId()****YOsControl****oscontrol**→**hardwareId()****oscontrol.get\_hardwareId()**

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the OS control (ex: `RELAYLO1-123456.relay1`)

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

**oscontrol**→**get\_logicalName()**

**YOsControl**

**oscontrol**→**logicalName()**

**oscontrol.get\_logicalName()**

---

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 on-line.

**Returns :**

an instance of YModule

**oscontrol→get\_shutdownCountdown()**

**YOsControl**

**oscontrol→shutdownCountdown()**

**oscontrol.get\_shutdownCountdown()**

---

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()**

**YOsControl**

**oscontrol**→**setLogicalName()**

**oscontrol.set\_logicalName()**

---

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( 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

---

**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.



## 3.30. 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');
c++	#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, in Watt, as a floating point number.

#### power→get\_currentValue()

Returns the current value of the electrical power, in Watt, as a floating point number.

#### 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.

**power→get\_hardwareId()**

Returns the unique hardware identifier of the electrical power sensor in the form `SERIAL.FUNCTIONID`.

**power→get\_highestValue()**

Returns the maximal value observed for the electrical power since the device was started.

**power→get\_logFrequency()**

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

**power→get\_logicalName()**

Returns the logical name of the electrical power sensor.

**power→get\_lowestValue()**

Returns the minimal value observed for the electrical power since the device was started.

**power→get\_meter()**

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

**power→get\_meterTimer()**

Returns the elapsed time since last energy counter reset, in seconds.

**power→get\_module()**

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

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

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

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

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

**power→get\_reportFrequency()**

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

**power→get\_resolution()**

Returns the resolution of the measured values.

**power→get\_unit()**

Returns the measuring unit for the electrical power.

**power→get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**power→isOnline()**

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

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

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

**power→load(msValidity)**

Preloads the electrical power sensor cache with a specified validity duration.

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

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

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

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

**power→nextPower()**

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

**power→registerTimedReportCallback(callback)**

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

**power→registerValueCallback(callback)**

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.

**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.

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

Changes the timed value notification frequency for this function.

**power**→**set\_resolution(newval)**

Changes the resolution of the measured physical 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()**  
**yFirstPower()YPower.FirstPower()**

---

**YPower**

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)=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 electrical power sensor (ex:  
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

**power**→**get\_advertisedValue()**

**YPower**

**power**→**advertisedValue()**

**power.get\_advertisedValue()**

---

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()**

**YPower**

**power**→**currentRawValue()**

**power.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number.

```
def get_currentRawValue()
```

**Returns :**

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

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

---

**power**→**get\_currentValue()****YPower****power**→**currentValue()****power.get\_currentValue()**

---

Returns the current value of the electrical power, in Watt, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the electrical power, in Watt, as a floating point number

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

**power**→**get\_errorMessage()**

**YPower**

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

---

Returns the error message of the latest error with the electrical power 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 electrical power sensor object

---

**power**→**get\_errorType()****YPower****power**→**errorType()****power.get\_errorType()**

---

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

**power**→**get\_friendlyName()**

**YPower**

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

---

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 example: `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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the electrical power sensor (ex: `RELAYLO1-123456.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 since the device was started.

```
def get_highestValue( )
```

**Returns :**

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

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 since the device was started.

```
def get_lowestValue() ( )
```

**Returns :**

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

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 on-line.

**Returns :**

an instance of YModule



---

**power**→**get\_recordedData()****YPower****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 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.

**power**→**get\_reportFrequency()**

**YPower**

**power**→**reportFrequency()**

**power.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`.

---

**power**→**get\_resolution()****YPower****power**→**resolution()****power.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`.

**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.

```
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.

**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.

```
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.



---

**power**→**set\_reportFrequency()****YPower****power**→**setReportFrequency()****power.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.

**power**→**set\_resolution()**

**YPower**

**power**→**setResolution()****power.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.

---

**power**→**set\_userData()****YPower****power**→**setUserData()****power.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

## 3.31. 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	<code>&lt;script type='text/javascript' src='yocto_pressure.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib');</code> <code>var YPressure = yoctolib.YPressure;</code>
php	<code>require_once('yocto_pressure.php');</code>
c++	<code>#include "yocto_pressure.h"</code>
m	<code>#import "yocto_pressure.h"</code>
pas	<code>uses yocto_pressure;</code>
vb	<code>yocto_pressure.vb</code>
cs	<code>yocto_pressure.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YPressure;</code>
py	<code>from yocto_pressure import *</code>

### 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 uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number.

#### **pressure→get\_currentValue()**

Returns the current value of the pressure, in millibar (hPa), as a floating point number.

#### **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`.

**pressure**→**get\_highestValue()**

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

**pressure**→**get\_logFrequency()**

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

**pressure**→**get\_logicalName()**

Returns the logical name of the pressure sensor.

**pressure**→**get\_lowestValue()**

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

**pressure**→**get\_module()**

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

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

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

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

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

**pressure**→**get\_reportFrequency()**

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

**pressure**→**get\_resolution()**

Returns the resolution of the measured values.

**pressure**→**get\_unit()**

Returns the measuring unit for the pressure.

**pressure**→**get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**pressure**→**isOnline()**

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

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

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

**pressure**→**load(msValidity)**

Preloads the pressure sensor cache with a specified validity duration.

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

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

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

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

**pressure**→**nextPressure()**

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

**pressure**→**registerTimedReportCallback(callback)**

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

**pressure**→**registerValueCallback(callback)**

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

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

Changes the recorded maximal value observed.

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

Changes the datalogger recording frequency for this function.

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

### 3. Reference

---

Changes the logical name of the pressure sensor.

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

Changes the recorded minimal value observed.

**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()**  
**pressure.calibrateFromPoints()****YPressure**

---

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` 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)=RELAYLO1-123456.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)=RELAYLO1-123456.relay1`)

---

**pressure**→**get\_advertisedValue()**

**YPressure**

**pressure**→**advertisedValue()**

**pressure.get\_advertisedValue()**

---

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()**

**YPressure**

**pressure**→**currentRawValue()**

**pressure.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number.

```
def get_currentRawValue()
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number

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

---

**pressure**→**get\_currentValue()****YPressure****pressure**→**currentValue()****pressure.get\_currentValue()**

---

Returns the current value of the pressure, in millibar (hPa), as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the pressure, in millibar (hPa), as a floating point number

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

**pressure**→**get\_errorMessage()**

**YPressure**

**pressure**→**errorMessage()**

**pressure**.**get\_errorMessage()**

---

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 example: `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`.



---

**pressure**→**get\_functionDescriptor()**

**YPressure**

**pressure**→**functionDescriptor()**

**pressure.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`.

**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\_hardwareId()****YPressure****pressure**→**hardwareId()****pressure.get\_hardwareId()**

---

Returns the unique hardware identifier of the pressure 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 pressure sensor (for example `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the pressure sensor (ex: `RELAYLO1-123456.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 since the device was started.

```
def get_highestValue( )
```

**Returns :**

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

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

---

**pressure**→**get\_logFrequency()****YPressure****pressure**→**logFrequency()****pressure.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`.

**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 since the device was started.

```
def get_lowestValue() ( )
```

**Returns :**

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

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 on-line.

**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 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.

**pressure**→**get\_reportFrequency()**

**YPressure**

**pressure**→**reportFrequency()**

**pressure.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.

---

**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()**

**YPressure**

**pressure**→**setHighestValue()**

**pressure.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.

---

**pressure**→**set\_logFrequency()****YPressure****pressure**→**setLogFrequency()****pressure.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.

**pressure**→**set\_logicalName()**

**YPressure**

**pressure**→**setLogicalName()**

**pressure.set\_logicalName()**

---

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.

```
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.

---

**pressure**→**set\_reportFrequency()****YPressure****pressure**→**setReportFrequency()****pressure.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.



---

**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

## 3.32. PwmInput 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_pwminput.js'></script>
nodejs	var yoctolib = require('yoctolib'); var YPwmInput = yoctolib.YPwmInput;
php	require_once('yocto_pwminput.php');
c++	#include "yocto_pwminput.h"
m	#import "yocto_pwminput.h"
pas	uses yocto_pwminput;
vb	yocto_pwminput.vb
cs	yocto_pwminput.cs
java	import com.yoctopuce.YoctoAPI.YPwmInput;
py	from yocto_pwminput import *

### Global functions

#### yFindPwmInput(func)

Retrieves a voltage sensor for a given identifier.

#### yFirstPwmInput()

Starts the enumeration of voltage sensors currently accessible.

### YPwmInput methods

#### pwminput→calibrateFromPoints(rawValues, refValues)

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

#### pwminput→describe()

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

#### pwminput→get\_advertisedValue()

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

#### pwminput→get\_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.

#### pwminput→get\_currentValue()

Returns the current value of PwmInput feature as a floating point number.

#### pwminput→get\_dutyCycle()

Returns the PWM duty cycle, in per cents.

#### pwminput→get\_errorMessage()

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

#### pwminput→get\_errorType()

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

#### pwminput→get\_frequency()

Returns the PWM frequency in Hz.

#### pwminput→get\_friendlyName()

Returns a global identifier of the voltage sensor in the format `MODULE_NAME . FUNCTION_NAME`.

#### pwminput→get\_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**pwminput→get\_functionId()**

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

**pwminput→get\_hardwareId()**

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

**pwminput→get\_highestValue()**

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

**pwminput→get\_logFrequency()**

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

**pwminput→get\_logicalName()**

Returns the logical name of the voltage sensor.

**pwminput→get\_lowestValue()**

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

**pwminput→get\_module()**

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

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

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

**pwminput→get\_period()**

Returns the PWM period in milliseconds.

**pwminput→get\_pulseCounter()**

Returns the pulse counter value.

**pwminput→get\_pulseDuration()**

Returns the PWM pulse length in milliseconds, as a floating point number.

**pwminput→get\_pulseTimer()**

Returns the timer of the pulses counter (ms)

**pwminput→get\_pwmReportMode()**

Returns the parameter (frequency/duty cycle, pulse width, edges count) returned by the get\_currentValue function and callbacks.

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

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

**pwminput→get\_reportFrequency()**

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

**pwminput→get\_resolution()**

Returns the resolution of the measured values.

**pwminput→get\_unit()**

Returns the measuring unit for the values returned by get\_currentValue and callbacks.

**pwminput→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set\_userData.

**pwminput→isOnline()**

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

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

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

**pwminput→load(msValidity)**

Preloads the voltage sensor cache with a specified validity duration.

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

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

**`pwminput→load_async(msValidity, callback, context)`**

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

**`pwminput→nextPwmInput()`**

Continues the enumeration of voltage sensors started using `yFirstPwmInput()`.

**`pwminput→registerTimedReportCallback(callback)`**

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

**`pwminput→registerValueCallback(callback)`**

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

**`pwminput→resetCounter()`**

Returns the pulse counter value as well as his timer

**`pwminput→set_highestValue(newval)`**

Changes the recorded maximal value observed.

**`pwminput→set_logFrequency(newval)`**

Changes the datalogger recording frequency for this function.

**`pwminput→set_logicalName(newval)`**

Changes the logical name of the voltage sensor.

**`pwminput→set_lowestValue(newval)`**

Changes the recorded minimal value observed.

**`pwminput→set_pwmReportMode(newval)`**

Modify the parameter type(frequency/duty cycle, pulse width ou edge count) returned by the `get_currentValue` function and callbacks.

**`pwminput→set_reportFrequency(newval)`**

Changes the timed value notification frequency for this function.

**`pwminput→set_resolution(newval)`**

Changes the resolution of the measured physical values.

**`pwminput→set_userData(data)`**

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

**`pwminput→wait_async(callback, context)`**

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

## YPwmInput.FindPwmInput() yFindPwmInput()YPwmInput.FindPwmInput()

YPwmInput

Retrieves a voltage sensor for a given identifier.

```
def FindPwmInput( 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 `YPwmInput.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 `YPwmInput` object allowing you to drive the voltage sensor.

---

**YPwmInput.FirstPwmInput()  
yFirstPwmInput()YPwmInput.FirstPwmInput()**

---

**YPwmInput**

Starts the enumeration of voltage sensors currently accessible.

```
def FirstPwmInput( )
```

Use the method `YPwmInput.nextPwmInput( )` to iterate on next voltage sensors.

**Returns :**

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

**pwminput**→**calibrateFromPoints()**  
**pwminput.calibrateFromPoints()****YPwmInput**

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.



---

**pwminput**→**describe()****pwminput.describe()****YPwmInput**

---

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` 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)=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`)

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

---

**YPwmInput**

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`.

---

**pwminput→get\_currentRawValue()****YPwmInput****pwminput→currentRawValue()****pwminput.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.

```
def get_currentRawValue( )
```

**Returns :**

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

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

**pwminput→get\_currentValue()**

**YPwmInput**

**pwminput→currentValue()**

**pwminput.get\_currentValue()**

---

Returns the current value of PwmInput feature as a floating point number.

```
def get_currentValue( )
```

Depending on the pwmReportMode setting, this can be the frequency, in Hz, the duty cycle in % or the pulse length.

**Returns :**

a floating point number corresponding to the current value of PwmInput feature as a floating point number

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

---

**pwminput**→**get\_dutyCycle()****YPwmInput****pwminput**→**dutyCycle()****pwminput.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`.

`pwminput→get_errorMessage()`

**YPwmInput**

`pwminput→errorMessage()`

`pwminput.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

---

**pwminput**→**get\_errorType()****YPwmInput****pwminput**→**errorType()****pwminput.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

**pwminput**→**get\_frequency()**

**YPwmInput**

**pwminput**→**frequency()****pwminput.get\_frequency()**

---

Returns the PWM frequency in Hz.

```
def get_frequency( )
```

**Returns :**

a floating point number corresponding to the PWM frequency in Hz

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



---

**pwminput**→**get\_friendlyName()****YPwmInput****pwminput**→**friendlyName()****pwminput.get\_friendlyName()**

---

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 example: `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`.

`pwminput`→`get_functionDescriptor()`

**YPwmInput**

`pwminput`→`functionDescriptor()`

`pwminput.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`.

---

**pwminput**→**get\_functionId()****YPwmInput****pwminput**→**functionId()****pwminput.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`.

**pwminput**→**get\_hardwareId()**

**YPwmInput**

**pwminput**→**hardwareId()****pwminput.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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the voltage sensor (ex: `RELAYLO1-123456.relay1`)

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

---

**pwminput→get\_highestValue()****YPwmInput****pwminput→highestValue()****pwminput.get\_highestValue()**

---

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

```
def get_highestValue( )
```

**Returns :**

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

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

**pwminput→get\_logFrequency()**

**YPwmInput**

**pwminput→logFrequency()**

**pwminput.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`.

---

**pwminput→get\_logicalName()**

**YPwmInput**

**pwminput→logicalName()**

**pwminput.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`.

**pwminput→get\_lowestValue()**

**YPwmInput**

**pwminput→lowestValue()**

**pwminput.get\_lowestValue()**

---

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

```
def get_lowestValue( )
```

**Returns :**

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

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



---

**pwminput**→**get\_module()****YPwmInput****pwminput**→**module()****pwminput.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`

`pwminput`→`get_period()`

YPwmInput

`pwminput`→`period()``pwminput.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`.

---

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

---

**YPwmInput**

Returns the pulse counter value.

```
def get_pulseCounter( )
```

Actually that counter is incremented twice per period. That counter is limited to 1 billions

**Returns :**

an integer corresponding to the pulse counter value

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

**pwminput→get\_pulseDuration()**  
**pwminput→pulseDuration()**  
**pwminput.get\_pulseDuration()**

---

**YPwmInput**

Returns the PWM pulse length in milliseconds, as a floating point number.

```
def get_pulseDuration( )
```

**Returns :**

a floating point number corresponding to the PWM pulse length in milliseconds, as a floating point number

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

---

**pwminput**→**get\_pulseTimer()****YPwmInput****pwminput**→**pulseTimer()****pwminput.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`.

**pwminput→get\_pwmReportMode()**

**YPwmInput**

**pwminput→pwmReportMode()**

**pwminput.get\_pwmReportMode()**

Returns the parameter (frequency/duty cycle, pulse width, edges count) returned by the `get_currentValue` function and callbacks.

```
def get_pwmReportMode( )
```

Attention

**Returns :**

a value among `Y_PWMREPORTMODE_PWM_DUTYCYCLE`, `Y_PWMREPORTMODE_PWM_FREQUENCY`, `Y_PWMREPORTMODE_PWM_PULSEDURATION` and `Y_PWMREPORTMODE_PWM_EDGECOUNT` corresponding to the parameter (frequency/duty cycle, pulse width, edges count) returned by the `get_currentValue` function and callbacks

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

---

**pwminput**→**get\_recordedData()****YPwmInput****pwminput**→**recordedData()****pwminput.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.

`pwminput→get_reportFrequency()`

**YPwmInput**

`pwminput→reportFrequency()`

`pwminput.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`.



---

**pwminput**→**get\_resolution()****YPwmInput****pwminput**→**resolution()****pwminput.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`.

**pwminput**→**get\_unit()**

**YPwmInput**

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

---

Returns the measuring unit for the values returned by `get_currentValue` and callbacks.

```
def get_unit( )
```

That unit will change according to the `pwmReportMode` settings.

**Returns :**

a string corresponding to the measuring unit for the values returned by `get_currentValue` and callbacks

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

---

**pwminput**→**get\_userData()****YPwmInput****pwminput**→**userData()****pwminput.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.

**pwminput**→**isOnline()****pwminput.isOnline()**

**YPwmInput**

---

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

---

**pwminput**→**load()****pwminput.load()****YPwmInput**

---

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.

**pwminput**→**loadCalibrationPoints()**  
**pwminput.loadCalibrationPoints()****YPwmInput**

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.

---

**pwminput**→**nextPwmInput()**  
**pwminput.nextPwmInput()**

---

**YPwmInput**

Continues the enumeration of voltage sensors started using `yFirstPwmInput()`.

```
def nextPwmInput( )
```

**Returns :**

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

**pwminput**→**registerTimedReportCallback()**  
**pwminput.registerTimedReportCallback()****YPwmInput**

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.



---

**pwminput→registerValueCallback()  
pwminput.registerValueCallback()**

---

**YPwmInput**

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.

**pwminput**→**resetCounter()****pwminput.resetCounter()**

**YPwmInput**

---

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.

---

**pwminput→set\_highestValue()**

**YPwmInput**

**pwminput→setHighestValue()**

**pwminput.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.

**pwminput→set\_logFrequency()**

**YPwmInput**

**pwminput→setLogFrequency()**

**pwminput.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.

---

**pwminput**→**set\_logicalName()****YPwmInput****pwminput**→**setLogicalName()****pwminput.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.

`pwminput`→`set_lowestValue()`  
`pwminput`→`setLowestValue()`  
`pwminput.set_lowestValue()`

---

YPwmInput

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.

---

**pwminput**→**set\_pwmReportMode()****YPwmInput****pwminput**→**setPwmReportMode()****pwminput.set\_pwmReportMode()**

---

Modify the parameter type(frequency/duty cycle, pulse width ou edge count) returned by the `get_currentValue` function and callbacks.

```
def set_pwmReportMode( newval)
```

The edge count value will be limited to the 6 lowest digit, for values greater than one million, use `get_pulseCounter()`.

**Parameters :**

```
newval a value among Y_PWMREPORTMODE_PWM_DUTYCYCLE,  
Y_PWMREPORTMODE_PWM_FREQUENCY,  
Y_PWMREPORTMODE_PWM_PULSEDURATION and  
Y_PWMREPORTMODE_PWM_EDGECOUNT
```

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

`pwminput`→`set_reportFrequency()`  
`pwminput`→`setReportFrequency()`  
`pwminput.set_reportFrequency()`

YPwmInput

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.



---

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

---

**YPwmInput**

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.

**pwminput**→**set\_userData()**

**YPwmInput**

**pwminput**→**setUserData()****pwminput.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

### 3.33. 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;
php	require_once('yocto_pwmoutput.php');
c++	#include "yocto_pwmoutput.h"
m	#import "yocto_pwmoutput.h"
pas	uses yocto_pwmoutput;
vb	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()

### 3. Reference

Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.
<b><code>pwmoutput→get_functionId()</code></b>
Returns the hardware identifier of the PWM, without reference to the module.
<b><code>pwmoutput→get_hardwareId()</code></b>
Returns the unique hardware identifier of the PWM in the form <code>SERIAL.FUNCTIONID</code> .
<b><code>pwmoutput→get_logicalName()</code></b>
Returns the logical name of the PWM.
<b><code>pwmoutput→get_module()</code></b>
Gets the <code>YModule</code> object for the device on which the function is located.
<b><code>pwmoutput→get_module_async(callback, context)</code></b>
Gets the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<b><code>pwmoutput→get_period()</code></b>
Returns the PWM period in milliseconds.
<b><code>pwmoutput→get_pulseDuration()</code></b>
Returns the PWM pulse length in milliseconds, as a floating point number.
<b><code>pwmoutput→get_userData()</code></b>
Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<b><code>pwmoutput→isOnline()</code></b>
Checks if the PWM is currently reachable, without raising any error.
<b><code>pwmoutput→isOnline_async(callback, context)</code></b>
Checks if the PWM is currently reachable, without raising any error (asynchronous version).
<b><code>pwmoutput→load(msValidity)</code></b>
Preloads the PWM cache with a specified validity duration.
<b><code>pwmoutput→load_async(msValidity, callback, context)</code></b>
Preloads the PWM cache with a specified validity duration (asynchronous version).
<b><code>pwmoutput→nextPwmOutput()</code></b>
Continues the enumeration of PWMs started using <code>yFirstPwmOutput()</code> .
<b><code>pwmoutput→pulseDurationMove(ms_target, ms_duration)</code></b>
Performs a smooth transistion of the pulse duration toward a given value.
<b><code>pwmoutput→registerValueCallback(callback)</code></b>
Registers the callback function that is invoked on every change of advertised value.
<b><code>pwmoutput→set_dutyCycle(newval)</code></b>
Changes the PWM duty cycle, in per cents.
<b><code>pwmoutput→set_dutyCycleAtPowerOn(newval)</code></b>
Changes the PWM duty cycle at device power on.
<b><code>pwmoutput→set_enabled(newval)</code></b>
Stops or starts the PWM.
<b><code>pwmoutput→set_enabledAtPowerOn(newval)</code></b>
Changes the state of the PWM at device power on.
<b><code>pwmoutput→set_frequency(newval)</code></b>
Changes the PWM frequency.
<b><code>pwmoutput→set_logicalName(newval)</code></b>
Changes the logical name of the PWM.
<b><code>pwmoutput→set_period(newval)</code></b>
Changes the PWM period in milliseconds.

**pwmoutput→set\_pulseDuration(newval)**

Changes the PWM pulse length, in milliseconds.

**pwmoutput→set\_userData(data)**

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)=RELAYLO1-123456.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)=RELAYLO1-123456.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()`

**YPwmOutput**

`pwmoutput→advertisedValue()`

`pwmoutput.get_advertisedValue()`

---

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()`

**YPwmOutput**

`pwmoutput→dutyCycleAtPowerOn()`

`pwmoutput.get_dutyCycleAtPowerOn()`

---

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_DUTYCYCLEATPOWERON_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()`

**YPwmOutput**

`pwmoutput→enabledAtPowerOn()`

`pwmoutput.get_enabledAtPowerOn()`

---

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 :**

a floating point number corresponding to the PWM frequency in Hz

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

**pwmoutput→get\_friendlyName()**

**YPwmOutput**

**pwmoutput→friendlyName()**

**pwmoutput.get\_friendlyName()**

---

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 example: `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()**

**YPwmOutput**

**pwmoutput**→**functionDescriptor()**

**pwmoutput.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`.

`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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the PWM (ex: `RELAYLO1-123456.relay1`)

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

`pwmoutput→get_logicalName()`

**YPwmOutput**

`pwmoutput→logicalName()`

`pwmoutput.get_logicalName()`

---

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 on-line.

**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()**

**YPwmOutput**

**pwmoutput→pulseDuration()**

**pwmoutput.get\_pulseDuration()**

---

Returns the PWM pulse length in milliseconds, as a floating point number.

```
def get_pulseDuration( )
```

**Returns :**

a floating point number corresponding to the PWM pulse length in milliseconds, as a floating point number

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 transistion 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()**

**YPwmOutput**

**pwmoutput→setDutyCycle()**

**pwmoutput.set\_dutyCycle()**

---

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** a floating point number 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()****YPwmOutput****pwmoutput**→**setLogicalName()****pwmoutput.set\_logicalName()**

---

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 in milliseconds.

```
def set_period( newval)
```

**Parameters :**

**newval** a floating point number corresponding to the PWM period in milliseconds

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**pwmoutput→set\_pulseDuration()**  
**pwmoutput→setPulseDuration()**  
**pwmoutput.set\_pulseDuration()**

---

**YPwmOutput**

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



## 3.34. 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).

### 3. Reference

#### **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(data)**

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` 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 source (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

**pwmpowersource→get\_advertisedValue()**

**YPwmPowerSource**

**pwmpowersource→advertisedValue()**

**pwmpowersource.get\_advertisedValue()**

---

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()`

**YPwmPowerSource**

`pwmpowersource→errorType()`

`pwmpowersource.get_errorType()`

---

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()****YPwmPowerSource****pwmpowersource→friendlyName()****pwmpowersource.get\_friendlyName()**

---

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 example: `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()`

`YPwmPowerSource`

`pwmpowersource`→`hardwareId()`

`pwmpowersource.get_hardwareId()`

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the voltage source (ex: `RELAYLO1-123456.relay1`)

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

---

`pwmpowersource`→`get_logicalName()`

**YPwmPowerSource**

`pwmpowersource`→`logicalName()`

`pwmpowersource.get_logicalName()`

---

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()**

**YPwmPowerSource**

**pwmpowersource→module()**

**pwmpowersource.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

---

`pwmpowersource`→`get_powerMode()`

**YPwmPowerSource**

`pwmpowersource`→`powerMode()`

`pwmpowersource.get_powerMode()`

---

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()`

**YPwmPowerSource**

`pwmpowersource→userData()`

`pwmpowersource.getUserData()`

---

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

```
def getUserData( )
```

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()****YPwmPowerSource****pwmpowersource**→**setLogicalName()****pwmpowersource.set\_logicalName()**

---

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()**

**YPwmPowerSource**

**pwmpowersource→setUserData()**

**pwmpowersource.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

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

### 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, in units, as a floating point number.

#### qt→get\_currentValue()

Returns the current value of the value, in units, as a floating point number.

#### 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.

**qt→get\_highestValue()**

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

**qt→get\_logFrequency()**

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

**qt→get\_logicalName()**

Returns the logical name of the quaternion component.

**qt→get\_lowestValue()**

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

**qt→get\_module()**

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

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

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

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

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

**qt→get\_reportFrequency()**

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

**qt→get\_resolution()**

Returns the resolution of the measured values.

**qt→get\_unit()**

Returns the measuring unit for the value.

**qt→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set\_userData.

**qt→isOnline()**

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

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

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

**qt→load(msValidity)**

Preloads the quaternion component cache with a specified validity duration.

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

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

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

Preloads the quaternion component cache with a specified validity duration (asynchronous version).

**qt→nextQt()**

Continues the enumeration of quaternion components started using yFirstQt().

**qt→registerTimedReportCallback(callback)**

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

**qt→registerValueCallback(callback)**

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

**qt→set\_highestValue(newval)**

Changes the recorded maximal value observed.

**qt→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**qt→set\_logicalName(newval)**

### 3. Reference

---

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() yFirstQt()YQt.FirstQt()

---

YQt

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)=RELAYLO1-123456.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)=RELAYLO1-123456.relay1`)

---

**qt**→**get\_advertisedValue()****YQt****qt**→**advertisedValue()****qt.get\_advertisedValue()**

---

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, in units, as a floating point number.

```
def get_currentRawValue( )
```

**Returns :**

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

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



---

**qt**→**get\_currentValue()****YQt****qt**→**currentValue()****qt.get\_currentValue()**

---

Returns the current value of the value, in units, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the value, in units, as a floating point number

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 example: `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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the quaternion component (ex: `RELAYLO1-123456.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()****YQt****qt**→**logFrequency()****qt.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`.

**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()****YQt****qt**→**lowestValue()****qt.get\_lowestValue()**

---

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 on-line.

**Returns :**

an instance of YModule

---

**qt**→**get\_recordedData()**

YQt

**qt**→**recordedData()****qt.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.

**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()**

**YQt**

**qt**→**unit()****qt.get\_unit()**

---

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()**

YQt

**qt→userdata()qt.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.

**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()**

YQt

**qt**→**setHighestValue()****qt.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.



---

**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()**

YQt

**qt**→**setLowestValue()****qt.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.

**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()**

YQt

**qt**→**setResolution()****qt.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.

**qt**→**set\_userdata()**

**YQt**

**qt**→**setUserData()****qt.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

## 3.36. Real Time Clock function interface

The RealTimeClock function maintains and provides current date and time, even accross 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	<code>&lt;script type='text/javascript' src='yocto_realtimelock.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib'); var YRealTimeClock = yoctolib.YRealTimeClock;</code>
php	<code>require_once('yocto_realtimelock.php');</code>
c++	<code>#include "yocto_realtimelock.h"</code>
m	<code>#import "yocto_realtimelock.h"</code>
pas	<code>uses yocto_realtimelock;</code>
vb	<code>yocto_realtimelock.vb</code>
cs	<code>yocto_realtimelock.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YRealTimeClock;</code>
py	<code>from yocto_realtimelock import *</code>

### Global functions

#### **yFindRealTimeClock(func)**

Retrieves a clock for a given identifier.

#### **yFirstRealTimeClock()**

Starts the enumeration of clocks currently accessible.

### YRealTimeClock methods

#### **realtimelock→describe()**

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

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

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

#### **realtimelock→get\_dateTime()**

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

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

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

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

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

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

Returns a global identifier of the clock in the format `MODULE_NAME . FUNCTION_NAME`.

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

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

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

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

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

Returns the unique hardware identifier of the clock in the form `SERIAL . FUNCTIONID`.

#### **realtimelock→get\_logicalName()**

Returns the logical name of the clock.

#### **realtimelock→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()**  
**yFirstRealTimeClock()**  
**YRealTimeClock.FirstRealTimeClock()**

---

**YRealTimeClock**

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.

**realtimelock→describe()realtimelock.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()

YRealTimeClock

realtimeclock→advertisedValue()

realtimeclock.get\_advertisedValue()

---

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**→**get\_dateTime()****YRealTimeClock****realtimeclock**→**dateTime()****realtimeclock.get\_dateTime()**

---

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

```
def get_dateTime( )
```

**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()**

**YRealTimeClock**

**realtimeclock→errorMessage()**

**realtimeclock.get\_errorMessage()**

---

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()**

**YRealTimeClock**

**realtimeclock→friendlyName()**

**realtimeclock.get\_friendlyName()**

---

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 example: `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()****YRealTimeClock****realtimeclock→functionDescriptor()****realtimeclock.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`.

`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`.

---

**realtimelock**→**get\_hardwareId()****YRealTimeClock****realtimelock**→**hardwareId()****realtimelock.get\_hardwareId()**

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the clock (ex: `RELAYLO1-123456.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 on-line.

**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\_TIMESET\_FALSE or Y\_TIMESET\_TRUE, according to true if the clock has been set, and false otherwise

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

---

**realtimeclock**→**get\_unixTime()****YRealTimeClock****realtimeclock**→**unixTime()****realtimeclock.get\_unixTime()**

---

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()**

**YRealTimeClock**

**realtimeclock→userData()**

**realtimeclock.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.



---

**realtimeclock**→**get\_utcOffset()****YRealTimeClock****realtimeclock**→**utcOffset()****realtimeclock.get\_utcOffset()**

---

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()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.

---

**realtimedclock**→**registerValueCallback()**  
**realtimedclock.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 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



---

**realtimeclock**→**set\_utcOffset()****YRealTimeClock****realtimeclock**→**setUtcOffset()****realtimeclock.set\_utcOffset()**

---

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.

## 3.37. 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	<code>&lt;script type='text/javascript' src='yocto_refframe.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib'); var YRefFrame = yoctolib.YRefFrame;</code>
php	<code>require_once('yocto_refframe.php');</code>
c++	<code>#include "yocto_refframe.h"</code>
m	<code>#import "yocto_refframe.h"</code>
pas	<code>uses yocto_refframe;</code>
vb	<code>yocto_refframe.vb</code>
cs	<code>yocto_refframe.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YRefFrame;</code>
py	<code>from yocto_refframe import *</code>

### 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.

**refframe→get\_errorMessage()**

Returns the error message of the latest error with the reference frame.

**refframe→get\_errorType()**

Returns the numerical error code of the latest error with the reference frame.

**refframe→get\_friendlyName()**

Returns a global identifier of the reference frame in the format `MODULE_NAME . FUNCTION_NAME`.

**refframe→get\_functionDescriptor()**

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

**refframe→get\_functionId()**

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

**refframe→get\_hardwareId()**

Returns the unique hardware identifier of the reference frame in the form `SERIAL . FUNCTIONID`.

**refframe→get\_logicalName()**

Returns the logical name of the reference frame.

**refframe→get\_module()**

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

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

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

**refframe→get\_mountOrientation()**

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.

**refframe→get\_mountPosition()**

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.

**refframe→get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**refframe→isOnline()**

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

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

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

**refframe→load(msValidity)**

Preloads the reference frame cache with a specified validity duration.

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

Preloads the reference frame cache with a specified validity duration (asynchronous version).

**refframe→more3DCalibration()**

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

**refframe→nextRefFrame()**

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

**refframe→registerValueCallback(callback)**

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

**refframe→save3DCalibration()**

Applies the sensors tridimensional calibration parameters that have just been computed.

**refframe→set\_bearing(newval)**

Changes the reference bearing used by the compass.

**refframe→set\_logicalName(newval)**

### 3. Reference

---

Changes the logical name of the reference frame.

**reframe**→**set\_mountPosition**(**position**, **orientation**)

Changes the compass and tilt sensor frame of reference.

**reframe**→**set\_userData**(**data**)

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

**reframe**→**start3DCalibration**()

Initiates the sensors tridimensional calibration process.

**reframe**→**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()****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)=RELAYLO1-123456.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)=RELAYLO1-123456.relay1`)



---

**refframe→get\_3DCalibrationHint()****YRefFrame****refframe→3DCalibrationHint()****refframe.get\_3DCalibrationHint()**

---

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

```
def get_3DCalibrationHint( )
```

**Returns :**

a character string.

**refframe**→**get\_3DCalibrationLogMsg()**  
**refframe**→**3DCalibrationLogMsg()**  
**refframe.get\_3DCalibrationLogMsg()**

---

**YRefFrame**

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()

YRefFrame

refframe→advertisedValue()

refframe.get\_advertisedValue()

---

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`.

`reframe→get_errorMessage()`  
`reframe→errorMessage()`  
`reframe.get_errorMessage()`

---

**YRefFrame**

Returns the error message of the latest error with the reference frame.

```
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 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 example: `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()****YRefFrame****refframe**→**functionDescriptor()****refframe.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`.

**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\_hardwareId()****YRefFrame****refframe**→**hardwareId()****refframe.get\_hardwareId()**

---

Returns the unique hardware identifier of the reference frame 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 reference frame (for example `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the reference frame (ex: `RELAYLO1-123456.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()**

**YRefFrame**

**refframe**→**mountOrientation()**

**refframe.get\_mountOrientation()**

---

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()****YRefFrame****refframe**→**mountPosition()****refframe.get\_mountPosition()**

---

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()****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()**

**YRefFrame**

**refframe**→**setLogicalName()**

**refframe.set\_logicalName()**

---

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.

## 3.38. 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	<code>&lt;script type='text/javascript' src='yocto_relay.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib'); var YRelay = yoctolib.YRelay;</code>
php	<code>require_once('yocto_relay.php');</code>
c++	<code>#include "yocto_relay.h"</code>
m	<code>#import "yocto_relay.h"</code>
pas	<code>uses yocto_relay;</code>
vb	<code>yocto_relay.vb</code>
cs	<code>yocto_relay.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YRelay;</code>
py	<code>from yocto_relay import *</code>

### 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`.

**relay→get\_logicalName()**

Returns the logical name of the relay.

**relay→get\_maxTimeOnStateA()**

Retourne the maximum time (ms) allowed for `$THEFUNCTIONS$` to stay in state A before automatically switching back in to B state.

**relay→get\_maxTimeOnStateB()**

Retourne the maximum time (ms) allowed for `$THEFUNCTIONS$` to stay in state B before automatically switching back in to A state.

**relay→get\_module()**

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

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

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

**relay→get\_output()**

Returns the output state of the relays, when used as a simple switch (single throw).

**relay→get\_pulseTimer()**

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

**relay→get\_state()**

Returns the state of the relays (A for the idle position, B for the active position).

**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).

**relay→get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**relay→isOnline()**

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

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

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

**relay→load(msValidity)**

Preloads the relay cache with a specified validity duration.

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

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

**relay→nextRelay()**

Continues the enumeration of relays started using `yFirstRelay()`.

**relay→pulse(ms\_duration)**

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

**relay→registerValueCallback(callback)**

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

**relay→set\_logicalName(newval)**

Changes the logical name of the relay.

**relay→set\_maxTimeOnStateA(newval)**

Sets the maximum time (ms) allowed for `$THEFUNCTIONS$` to stay in state A before automatically switching back in to B state.

**relay→set\_maxTimeOnStateB(newval)**

### 3. Reference

---

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 milliseconds

**ms\_duration** pulse duration, in milliseconds

**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)=RELAYLO1-123456.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)=RELAYLO1-123456.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.get\_errorMessage()**

---

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

```
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 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



---

**relay**→**get\_friendlyName()****YRelay****relay**→**friendlyName()****relay.get\_friendlyName()**

---

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 example: `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`.

---

**relay**→**get\_functionId()****YRelay****relay**→**functionId()****relay.get\_functionId()**

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the relay (ex: `RELAYLO1-123456.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()**

**YRelay**

**relay**→**maxTimeOnStateA()**

**relay.get\_maxTimeOnStateA()**

---

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()****YRelay****relay**→**maxTimeOnStateB()****relay.get\_maxTimeOnStateB()**

---

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 on-line.

**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_PULSETIMER_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 milliseconds

**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.

---

**relay**→**set\_logicalName()****YRelay****relay**→**setLogicalName()****relay.set\_logicalName()**

---

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()**

**YRelay**

**relay**→**setMaxTimeOnStateA()**

**relay.set\_maxTimeOnStateA()**

---

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()****YRelay****relay**→**setMaxTimeOnStateB()****relay.set\_maxTimeOnStateB()**

---

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()**

**YRelay**

**relay**→**setStateAtPowerOn()**

**relay.set\_stateAtPowerOn()**

---

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( 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. 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, in the specified unit, as a floating point number.

#### sensor→get\_currentValue()

Returns the current value of the measure, in the specified unit, as a floating point number.

#### 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()

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

**sensor**→**get\_highestValue()**

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

**sensor**→**get\_logFrequency()**

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

**sensor**→**get\_logicalName()**

Returns the logical name of the sensor.

**sensor**→**get\_lowestValue()**

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

**sensor**→**get\_module()**

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

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

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

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

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

**sensor**→**get\_reportFrequency()**

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

**sensor**→**get\_resolution()**

Returns the resolution of the measured values.

**sensor**→**get\_unit()**

Returns the measuring unit for the measure.

**sensor**→**get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**sensor**→**isOnline()**

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

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

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

**sensor**→**load(msValidity)**

Preloads the sensor cache with a specified validity duration.

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

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

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

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

**sensor**→**nextSensor()**

Continues the enumeration of sensors started using `yFirstSensor()`.

**sensor**→**registerTimedReportCallback(callback)**

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

**sensor**→**registerValueCallback(callback)**

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

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

Changes the recorded maximal value observed.

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

Changes the datalogger recording frequency for this function.

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

### 3. Reference

---

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` 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 sensor (ex: `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1`)



---

**sensor**→**get\_advertisedValue()****YSensor****sensor**→**advertisedValue()****sensor.get\_advertisedValue()**

---

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()**

**YSensor**

**sensor**→**currentRawValue()**

**sensor.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number.

```
def get_currentRawValue()
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number

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, in the specified unit, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the measure, in the specified unit, as a floating point number

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

**sensor**→**get\_errorMessage()**

**YSensor**

**sensor**→**errorMessage()****sensor.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()`**YSensor****sensor**→`errorType()`**sensor.get\_errorType()**

---

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 example: `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()****YSensor****sensor**→**functionDescriptor()****sensor.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`.

**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`.



---

**sensor**→**get\_hardwareId()****YSensor****sensor**→**hardwareId()****sensor.get\_hardwareId()**

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the sensor (ex: `RELAYLO1-123456.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 on-line.

**Returns :**

an instance of YModule

---

**sensor**→**get\_recordedData()****YSensor****sensor**→**recordedData()****sensor.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.

**sensor**→**get\_reportFrequency()**

**YSensor**

**sensor**→**reportFrequency()**

**sensor.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.



---

**sensor**→**get\_resolution()****YSensor****sensor**→**resolution()****sensor.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`.

**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.

**sensor**→**set\_highestValue()**

**YSensor**

**sensor**→**setHighestValue()****sensor.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.

---

**sensor**→**set\_logFrequency()****YSensor****sensor**→**setLogFrequency()****sensor.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.

**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.

---

**sensor**→**set\_lowestValue()****YSensor****sensor**→**setLowestValue()****sensor.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.

**sensor**→**set\_reportFrequency()**

**YSensor**

**sensor**→**setReportFrequency()**

**sensor.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.

---

**sensor**→**set\_resolution()****YSensor****sensor**→**setResolution()****sensor.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.

**sensor**→**set\_userData()**

**YSensor**

**sensor**→**setUserData()****sensor.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



## 3.40. SerialPort function interface

The SerialPort function interface allows you to fully drive a Yoctopuce serial port, to send and receive data, and to configure communication parameters (baud rate, bit count, parity, flow control and protocol). Note that Yoctopuce serial ports are not exposed as virtual COM ports. They are meant to be used in the same way as all Yoctopuce devices.

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

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

### Global functions

#### yFindSerialPort(func)

Retrieves a serial port for a given identifier.

#### yFirstSerialPort()

Starts the enumeration of serial ports currently accessible.

### YSerialPort methods

#### serialport→describe()

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

#### serialport→get\_CTS()

Read the level of the CTS line.

#### serialport→get\_advertisedValue()

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

#### serialport→get\_errCount()

Returns the total number of communication errors detected since last reset.

#### serialport→get\_errorMessage()

Returns the error message of the latest error with the serial port.

#### serialport→get\_errorType()

Returns the numerical error code of the latest error with the serial port.

#### serialport→get\_friendlyName()

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

#### serialport→get\_functionDescriptor()

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

#### serialport→get\_functionId()

Returns the hardware identifier of the serial port, without reference to the module.

#### serialport→get\_hardwareId()

Returns the unique hardware identifier of the serial port in the form SERIAL . FUNCTIONID.

#### serialport→get\_lastMsg()

	Returns the latest message fully received (for Line, Frame and Modbus protocols).
<b>serialport</b> → <b>get_logicalName()</b>	Returns the logical name of the serial port.
<b>serialport</b> → <b>get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>serialport</b> → <b>get_module_async(callback, context)</b>	Gets the YModule object for the device on which the function is located (asynchronous version).
<b>serialport</b> → <b>get_msgCount()</b>	Returns the total number of messages received since last reset.
<b>serialport</b> → <b>get_protocol()</b>	Returns the type of protocol used over the serial line, as a string.
<b>serialport</b> → <b>get_rxCount()</b>	Returns the total number of bytes received since last reset.
<b>serialport</b> → <b>get_serialMode()</b>	Returns the serial port communication parameters, as a string such as "9600,8N1".
<b>serialport</b> → <b>get_txCount()</b>	Returns the total number of bytes transmitted since last reset.
<b>serialport</b> → <b>get_userData()</b>	Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
<b>serialport</b> → <b>isOnline()</b>	Checks if the serial port is currently reachable, without raising any error.
<b>serialport</b> → <b>isOnline_async(callback, context)</b>	Checks if the serial port is currently reachable, without raising any error (asynchronous version).
<b>serialport</b> → <b>load(msValidity)</b>	Preloads the serial port cache with a specified validity duration.
<b>serialport</b> → <b>load_async(msValidity, callback, context)</b>	Preloads the serial port cache with a specified validity duration (asynchronous version).
<b>serialport</b> → <b>modbusReadBits(slaveNo, pduAddr, nBits)</b>	Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.
<b>serialport</b> → <b>modbusReadInputBits(slaveNo, pduAddr, nBits)</b>	Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.
<b>serialport</b> → <b>modbusReadInputRegisters(slaveNo, pduAddr, nWords)</b>	Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.
<b>serialport</b> → <b>modbusReadRegisters(slaveNo, pduAddr, nWords)</b>	Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.
<b>serialport</b> → <b>modbusWriteAndReadRegisters(slaveNo, pduWriteAddr, values, pduReadAddr, nReadWords)</b>	Sets several contiguous internal registers (holding registers) on a MODBUS serial device, then performs a contiguous read of a set of (possibly different) internal registers.
<b>serialport</b> → <b>modbusWriteBit(slaveNo, pduAddr, value)</b>	Sets a single internal bit (or coil) on a MODBUS serial device.
<b>serialport</b> → <b>modbusWriteBits(slaveNo, pduAddr, bits)</b>	Sets several contiguous internal bits (or coils) on a MODBUS serial device.
<b>serialport</b> → <b>modbusWriteRegister(slaveNo, pduAddr, value)</b>	Sets a single internal register (or holding register) on a MODBUS serial device.
<b>serialport</b> → <b>modbusWriteRegisters(slaveNo, pduAddr, values)</b>	

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

**serialport**→**nextSerialPort()**

Continues the enumeration of serial ports started using `yFirstSerialPort()`.

**serialport**→**queryLine(query, maxWait)**

Sends a text line query to the serial port, and reads the reply, if any.

**serialport**→**queryMODBUS(slaveNo, pduBytes)**

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

**serialport**→**readHex(nBytes)**

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

**serialport**→**readLine()**

Reads a single line (or message) from the receive buffer, starting at current stream position.

**serialport**→**readMessages(pattern, maxWait)**

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

**serialport**→**readStr(nChars)**

Reads data from the receive buffer as a string, starting at current stream position.

**serialport**→**read\_seek(rxCountVal)**

Changes the current internal stream position to the specified value.

**serialport**→**registerValueCallback(callback)**

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

**serialport**→**reset()**

Clears the serial port buffer and resets counters to zero.

**serialport**→**set\_RTS(val)**

Manually sets the state of the RTS line.

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

Changes the logical name of the serial port.

**serialport**→**set\_protocol(newval)**

Changes the type of protocol used over the serial line.

**serialport**→**set\_serialMode(newval)**

Changes the serial port communication parameters, with a string such as "9600,8N1".

**serialport**→**set\_userData(data)**

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

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

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

**serialport**→**writeArray(byteList)**

Sends a byte sequence (provided as a list of bytes) to the serial port.

**serialport**→**writeBin(buff)**

Sends a binary buffer to the serial port, as is.

**serialport**→**writeHex(hexString)**

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

**serialport**→**writeLine(text)**

Sends an ASCII string to the serial port, followed by a line break (CR LF).

**serialport**→**writeMODBUS(hexString)**

Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

**serialport**→**writeStr(text)**

### 3. Reference

---

Sends an ASCII string to the serial port, as is.

## YSerialPort.FindSerialPort() yFindSerialPort()YSerialPort.FindSerialPort()

YSerialPort

Retrieves a serial port for a given identifier.

```
def FindSerialPort( 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 serial port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSerialPort.isOnline()` to test if the serial port is indeed online at a given time. In case of ambiguity when looking for a serial 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 serial port

**Returns :**

a `YSerialPort` object allowing you to drive the serial port.

## YSerialPort.FirstSerialPort()

YSerialPort

### yFirstSerialPort()YSerialPort.FirstSerialPort()

---

Starts the enumeration of serial ports currently accessible.

```
def FirstSerialPort( )
```

Use the method `YSerialPort.nextSerialPort( )` to iterate on next serial ports.

**Returns :**

a pointer to a `YSerialPort` object, corresponding to the first serial port currently online, or a `null` pointer if there are none.

**serialport**→**describe()****serialport.describe()****YSerialPort**

Returns a short text that describes unambiguously the instance of the serial 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)=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 serial port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

**serialport**→**get\_CTS()**

**YSerialPort**

**serialport**→**CTS()****serialport.get\_CTS()**

---

Read the level of the CTS line.

```
def get_CTS( )
```

The CTS line is usually driven by the RTS signal of the connected serial device.

**Returns :**

1 if the CTS line is high, 0 if the CTS line is low.

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



---

**serialport**→**get\_advertisedValue()****YSerialPort****serialport**→**advertisedValue()****serialport.get\_advertisedValue()**

---

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

```
def get_advertisedValue( )
```

**Returns :**

a string corresponding to the current value of the serial port (no more than 6 characters).

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

**serialport**→**get\_errCount()**

**YSerialPort**

**serialport**→**errCount()****serialport.get\_errCount()**

---

Returns the total number of communication errors detected since last reset.

```
def get_errCount( )
```

**Returns :**

an integer corresponding to the total number of communication errors detected since last reset

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

---

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

---

**YSerialPort**

Returns the error message of the latest error with the serial 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 serial port object

**serialport**→**get\_errorType()**

**YSerialPort**

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

---

Returns the numerical error code of the latest error with the serial 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 serial port object

---

**serialport**→**get\_friendlyName()****YSerialPort****serialport**→**friendlyName()****serialport.get\_friendlyName()**

---

Returns a global identifier of the serial port in the format `MODULE_NAME.FUNCTION_NAME`.

```
def get_friendlyName( )
```

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

**Returns :**

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

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

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

---

**YSerialPort**

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`.

---

**serialport**→**get\_functionId()****YSerialPort****serialport**→**functionId()****serialport.get\_functionId()**

---

Returns the hardware identifier of the serial port, without reference to the module.

```
def get_functionId( )
```

For example `relay1`

**Returns :**

a string that identifies the serial port (ex: `relay1`)

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

**serialport**→**get\_hardwareId()**

**YSerialPort**

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

---

Returns the unique hardware identifier of the serial 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 serial port (for example `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the serial port (ex: `RELAYLO1-123456.relay1`)

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



---

**serialport**→**get\_lastMsg()****YSerialPort****serialport**→**lastMsg()****serialport.get\_lastMsg()**

---

Returns the latest message fully received (for Line, Frame and Modbus protocols).

```
def get_lastMsg( )
```

**Returns :**

a string corresponding to the latest message fully received (for Line, Frame and Modbus protocols)

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

**serialport**→**get\_logicalName()**

**YSerialPort**

**serialport**→**logicalName()**

**serialport.get\_logicalName()**

---

Returns the logical name of the serial port.

```
def get_logicalName( )
```

**Returns :**

a string corresponding to the logical name of the serial port.

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

---

**serialport**→**get\_module()****YSerialPort****serialport**→**module()****serialport.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`

**serialport**→**get\_msgCount()**

**YSerialPort**

**serialport**→**msgCount()****serialport.get\_msgCount()**

---

Returns the total number of messages received since last reset.

```
def get_msgCount( )
```

**Returns :**

an integer corresponding to the total number of messages received since last reset

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

---

**serialport**→**get\_protocol()****YSerialPort****serialport**→**protocol()****serialport.get\_protocol()**

---

Returns the type of protocol used over the serial line, as a string.

```
def get_protocol( )
```

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream.

**Returns :**

a string corresponding to the type of protocol used over the serial line, as a string

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

**serialport**→**get\_rxCount()**

**YSerialPort**

**serialport**→**rxCount()****serialport.get\_rxCount()**

---

Returns the total number of bytes received since last reset.

```
def get_rxCount( )
```

**Returns :**

an integer corresponding to the total number of bytes received since last reset

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

---

**serialport**→**get\_serialMode()****YSerialPort****serialport**→**serialMode()****serialport.get\_serialMode()**

---

Returns the serial port communication parameters, as a string such as "9600,8N1".

```
def get_serialMode( )
```

The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix is included if flow control is active: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance).

**Returns :**

a string corresponding to the serial port communication parameters, as a string such as "9600,8N1"

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

**serialport**→**get\_txCount()**

**YSerialPort**

**serialport**→**txCount()****serialport.get\_txCount()**

---

Returns the total number of bytes transmitted since last reset.

```
def get_txCount( )
```

**Returns :**

an integer corresponding to the total number of bytes transmitted since last reset

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



---

**serialport**→**get\_userData()****YSerialPort****serialport**→**userData()****serialport.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.

**serialport→isOnline()serialport.isOnline()**

**YSerialPort**

---

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

```
def isOnline( )
```

If there is a cached value for the serial 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 serial port.

**Returns :**

`true` if the serial port can be reached, and `false` otherwise

---

**serialport**→**load()****serialport.load()****YSerialPort**

---

Preloads the serial 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.

**serialport**→**modbusReadBits()**  
**serialport.modbusReadBits()****YSerialPort**

Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.

```
def modbusReadBits( slaveNo, pduAddr, nBits)
```

This method uses the MODBUS function code 0x01 (Read Coils).

**Parameters :**

- slaveNo** the address of the slave MODBUS device to query
- pduAddr** the relative address of the first bit/coil to read (zero-based)
- nBits** the number of bits/coils to read

**Returns :**

a vector of integers, each corresponding to one bit.

On failure, throws an exception or returns an empty array.

---

**serialport**→**modbusReadInputBits()**  
**serialport.modbusReadInputBits()**

---

**YSerialPort**

Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.

```
def modbusReadInputBits( slaveNo, pduAddr, nBits)
```

This method uses the MODBUS function code 0x02 (Read Discrete Inputs).

**Parameters :**

- slaveNo** the address of the slave MODBUS device to query
- pduAddr** the relative address of the first bit/input to read (zero-based)
- nBits** the number of bits/inputs to read

**Returns :**

a vector of integers, each corresponding to one bit.

On failure, throws an exception or returns an empty array.

**serialport**→**modbusReadInputRegisters()**  
**serialport.modbusReadInputRegisters()****YSerialPort**

Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.

```
def modbusReadInputRegisters( slaveNo, pduAddr, nWords)
```

This method uses the MODBUS function code 0x04 (Read Input Registers).

**Parameters :**

- slaveNo** the address of the slave MODBUS device to query
- pduAddr** the relative address of the first input register to read (zero-based)
- nWords** the number of input registers to read

**Returns :**

a vector of integers, each corresponding to one 16-bit input value.

On failure, throws an exception or returns an empty array.

---

**serialport**→**modbusReadRegisters()**  
**serialport.modbusReadRegisters()**

---

**YSerialPort**

Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

```
def modbusReadRegisters( slaveNo, pduAddr, nWords)
```

This method uses the MODBUS function code 0x03 (Read Holding Registers).

**Parameters :**

- slaveNo** the address of the slave MODBUS device to query
- pduAddr** the relative address of the first holding register to read (zero-based)
- nWords** the number of holding registers to read

**Returns :**

a vector of integers, each corresponding to one 16-bit register value.

On failure, throws an exception or returns an empty array.

**serialport**→**modbusWriteAndReadRegisters()**  
**serialport.modbusWriteAndReadRegisters()****YSerialPort**

Sets several contiguous internal registers (holding registers) on a MODBUS serial device, then performs a contiguous read of a set of (possibly different) internal registers.

```
def modbusWriteAndReadRegisters( slaveNo, pduWriteAddr, values, pduReadAddr, nReadWords)
```

This method uses the MODBUS function code 0x17 (Read/Write Multiple Registers).

**Parameters :**

- slaveNo** the address of the slave MODBUS device to drive
- pduWriteAddr** the relative address of the first internal register to set (zero-based)
- values** the vector of 16 bit values to set
- pduReadAddr** the relative address of the first internal register to read (zero-based)
- nReadWords** the number of 16 bit values to read

**Returns :**

a vector of integers, each corresponding to one 16-bit register value read.

On failure, throws an exception or returns an empty array.



---

**serialport**→**modbusWriteBit()**  
**serialport.modbusWriteBit()**

---

**YSerialPort**

Sets a single internal bit (or coil) on a MODBUS serial device.

```
def modbusWriteBit( slaveNo, pduAddr, value)
```

This method uses the MODBUS function code 0x05 (Write Single Coil).

**Parameters :**

- slaveNo** the address of the slave MODBUS device to drive
- pduAddr** the relative address of the bit/coil to set (zero-based)
- value** the value to set (0 for OFF state, non-zero for ON state)

**Returns :**

the number of bits/coils affected on the device (1)

On failure, throws an exception or returns zero.

**serialport**→**modbusWriteBits()**  
**serialport.modbusWriteBits()****YSerialPort**

Sets several contiguous internal bits (or coils) on a MODBUS serial device.

```
def modbusWriteBits( slaveNo, pduAddr, bits)
```

This method uses the MODBUS function code 0x0f (Write Multiple Coils).

**Parameters :**

- slaveNo** the address of the slave MODBUS device to drive
- pduAddr** the relative address of the first bit/coil to set (zero-based)
- bits** the vector of bits to be set (one integer per bit)

**Returns :**

the number of bits/coils affected on the device

On failure, throws an exception or returns zero.

---

**serialport**→**modbusWriteRegister()**  
**serialport.modbusWriteRegister()**

---

**YSerialPort**

Sets a single internal register (or holding register) on a MODBUS serial device.

```
def modbusWriteRegister( slaveNo, pduAddr, value)
```

This method uses the MODBUS function code 0x06 (Write Single Register).

**Parameters :**

- slaveNo** the address of the slave MODBUS device to drive
- pduAddr** the relative address of the register to set (zero-based)
- value** the 16 bit value to set

**Returns :**

the number of registers affected on the device (1)

On failure, throws an exception or returns zero.

**serialport**→**modbusWriteRegisters()**  
**serialport.modbusWriteRegisters()****YSerialPort**

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

```
def modbusWriteRegisters( slaveNo, pduAddr, values)
```

This method uses the MODBUS function code 0x10 (Write Multiple Registers).

**Parameters :**

- slaveNo** the address of the slave MODBUS device to drive
- pduAddr** the relative address of the first internal register to set (zero-based)
- values** the vector of 16 bit values to set

**Returns :**

the number of registers affected on the device

On failure, throws an exception or returns zero.

---

**serialport**→**nextSerialPort()****serialport.nextSerialPort()****YSerialPort**

---

Continues the enumeration of serial ports started using `yFirstSerialPort()`.

```
def nextSerialPort( )
```

**Returns :**

a pointer to a `YSerialPort` object, corresponding to a serial port currently online, or a `null` pointer if there are no more serial ports to enumerate.

**serialport**→**queryLine()****serialport.queryLine()****YSerialPort**

Sends a text line query to the serial port, and reads the reply, if any.

```
def queryLine( query, maxWait)
```

This function can only be used when the serial port is configured for 'Line' protocol.

**Parameters :**

**query** the line query to send (without CR/LF)

**maxWait** the maximum number of milliseconds to wait for a reply.

**Returns :**

the next text line received after sending the text query, as a string. Additional lines can be obtained by calling `readLine` or `readMessages`.

On failure, throws an exception or returns an empty array.

---

**serialport→queryMODBUS()**  
**serialport.queryMODBUS()**

---

**YSerialPort**

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

```
def queryMODBUS( slaveNo, pduBytes)
```

The message is the PDU, provided as a vector of bytes.

**Parameters :**

**slaveNo** the address of the slave MODBUS device to query

**pduBytes** the message to send (PDU), as a vector of bytes. The first byte of the PDU is the MODBUS function code.

**Returns :**

the received reply, as a vector of bytes.

On failure, throws an exception or returns an empty array (or a MODBUS error reply).

**serialport**→**readHex()****serialport.readHex()**

**YSerialPort**

---

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

```
def readHex( nBytes)
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

**Parameters :**

**nBytes** the maximum number of bytes to read

**Returns :**

a string with receive buffer contents, encoded in hexadecimal

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



---

**serialport**→**readLine()****serialport.readLine()****YSerialPort**

---

Reads a single line (or message) from the receive buffer, starting at current stream position.

```
def readLine( )
```

This function can only be used when the serial port is configured for a message protocol, such as 'Line' mode or MODBUS protocols. It does not work in plain stream modes, eg. 'Char' or 'Byte').

If data at current stream position is not available anymore in the receive buffer, the function returns the oldest available line and moves the stream position just after. If no new full line is received, the function returns an empty line.

**Returns :**

a string with a single line of text

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

**serialport**→**readMessages()**  
**serialport.readMessages()****YSerialPort**

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

```
def readMessages( pattern, maxWait)
```

This function can only be used when the serial port is configured for a message protocol, such as 'Line' mode or MODBUS protocols. It does not work in plain stream modes, eg. 'Char' or 'Byte', for which there is no "start" of message.

The search returns all messages matching the expression provided as argument in the buffer. If no matching message is found, the search waits for one up to the specified maximum timeout (in milliseconds).

**Parameters :**

**pattern** a limited regular expression describing the expected message format, or an empty string if all messages should be returned (no filtering). When using binary protocols, the format applies to the hexadecimal representation of the message.

**maxWait** the maximum number of milliseconds to wait for a message if none is found in the receive buffer.

**Returns :**

an array of strings containing the messages found, if any. Binary messages are converted to hexadecimal representation.

On failure, throws an exception or returns an empty array.

---

**serialport**→**readStr()****serialport.readStr()****YSerialPort**

---

Reads data from the receive buffer as a string, starting at current stream position.

```
def readStr( nChars)
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

**Parameters :**

**nChars** the maximum number of characters to read

**Returns :**

a string with receive buffer contents

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

**serialport**→**read\_seek()**serialport.read\_seek()

**YSerialPort**

---

Changes the current internal stream position to the specified value.

```
def read_seek( rxCountVal)
```

This function does not affect the device, it only changes the value stored in the YSerialPort object for the next read operations.

**Parameters :**

**rxCountVal** the absolute position index (value of rxCount) for next read operations.

**Returns :**

nothing.

---

**serialport**→**registerValueCallback()**  
**serialport.registerValueCallback()**

---

**YSerialPort**

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.

**serialport**→**reset()****serialport.reset()**

**YSerialPort**

---

Clears the serial port buffer and resets counters to zero.

```
def reset( )
```

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**serialport**→**set\_RTS()****YSerialPort****serialport**→**setRTS()****serialport.set\_RTS()**

---

Manually sets the state of the RTS line.

```
def set_RTS( val)
```

This function has no effect when hardware handshake is enabled, as the RTS line is driven automatically.

**Parameters :**

**val** 1 to turn RTS on, 0 to turn RTS off

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**serialport**→**set\_logicalName()****YSerialPort****serialport**→**setLogicalName()****serialport.set\_logicalName()**

---

Changes the logical name of the serial 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 serial port.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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



---

**serialport**→**set\_protocol()****YSerialPort****serialport**→**setProtocol()****serialport.set\_protocol()**

---

Changes the type of protocol used over the serial line.

```
def set_protocol( newval)
```

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream.

**Parameters :**

**newval** a string corresponding to the type of protocol used over the serial line

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**serialport**→**set\_serialMode()**

**YSerialPort**

**serialport**→**setSerialMode()**

**serialport.set\_serialMode()**

---

Changes the serial port communication parameters, with a string such as "9600,8N1".

```
def set_serialMode( newval)
```

The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix can be added to enable flow control: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance).

**Parameters :**

**newval** a string corresponding to the serial port communication parameters, with a string such as "9600,8N1"

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**serialport**→**set\_userData()****YSerialPort****serialport**→**setUserData()****serialport.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

---

**serialport**→**writeArray()****serialport.writeArray()**

**YSerialPort**

---

Sends a byte sequence (provided as a list of bytes) to the serial port.

```
def writeArray( byteList)
```

**Parameters :**

**byteList** a list of byte codes

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**serialport**→**writeBin()****serialport.writeBin()**

---

**YSerialPort**

Sends a binary buffer to the serial port, as is.

```
def writeBin( buff)
```

**Parameters :**

**buff** the binary buffer to send

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**serialport**→**writeHex()****serialport.writeHex()****YSerialPort**

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

```
def writeHex( hexString)
```

**Parameters :**

**hexString** a string of hexadecimal byte codes

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**serialport**→**writeLine()****serialport.writeLine()****YSerialPort**

---

Sends an ASCII string to the serial port, followed by a line break (CR LF).

```
def writeLine( text)
```

**Parameters :**

**text** the text string to send

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

**serialport**→**writeMODBUS()****serialport.writeMODBUS()**

**YSerialPort**

---

Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

```
def writeMODBUS( hexString)
```

The message must start with the slave address. The MODBUS CRC/LRC is automatically added by the function. This function does not wait for a reply.

**Parameters :**

**hexString** a hexadecimal message string, including device address but no CRC/LRC

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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



---

**serialport**→**writeStr()****serialport.writeStr()****YSerialPort**

---

Sends an ASCII string to the serial port, as is.

```
def writeStr( text)
```

**Parameters :**

**text** the text string to send

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

## 3.41. 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');
c++	#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() yFirstServo()YServo.FirstServo()**

---

**YServo**

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` 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)=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 servo (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.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()**  
**servo**→**enabled()****servo.get\_enabled()**

---

**YServo**

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()**

**YServo**

**servo**→**enabledAtPowerOn()**

**servo.get\_enabledAtPowerOn()**

---

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()****YServo****servo**→**errorMessage()****servo**.**get\_errorMessage()**

---

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

```
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 servo object

**servo**→**get\_errorType()**

**YServo**

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

---

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 example: `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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the servo (ex: `RELAYLO1-123456.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()**

**YServo**

**servo**→**module()****servo.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

---

**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()**

**YServo**

**servo**→**position()****servo.get\_position()**

---

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()****YServo****servo**→**positionAtPowerOn()****servo.get\_positionAtPowerOn()**

---

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.

---

**servo**→**set\_enabled()**  
**servo**→**setEnabled()****servo.set\_enabled()**

---

**YServo**

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()****YServo****servo**→**setNeutral()****servo.set\_neutral()**

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.



---

**servo**→**set\_position()****YServo****servo**→**setPosition()****servo.set\_position()**

---

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( 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

## 3.42. 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	<code>&lt;script type='text/javascript' src='yocto_temperature.js'&gt;&lt;/script&gt;</code>
nodejs	<code>var yoctolib = require('yoctolib'); var YTemperature = yoctolib.YTemperature;</code>
php	<code>require_once('yocto_temperature.php');</code>
c++	<code>#include "yocto_temperature.h"</code>
m	<code>#import "yocto_temperature.h"</code>
pas	<code>uses yocto_temperature;</code>
vb	<code>yocto_temperature.vb</code>
cs	<code>yocto_temperature.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YTemperature;</code>
py	<code>from yocto_temperature import *</code>

### 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, in Celsius, as a floating point number.

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

Returns the current value of the temperature, in Celsius, as a floating point number.

#### **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`.

### 3. Reference

#### **temperature**→**get\_highestValue()**

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

#### **temperature**→**get\_logFrequency()**

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

#### **temperature**→**get\_logicalName()**

Returns the logical name of the temperature sensor.

#### **temperature**→**get\_lowestValue()**

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

#### **temperature**→**get\_module()**

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

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

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

#### **temperature**→**get\_recordedData(startTime, endTime)**

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

#### **temperature**→**get\_reportFrequency()**

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

#### **temperature**→**get\_resolution()**

Returns the resolution of the measured values.

#### **temperature**→**get\_sensorType()**

Returns the temperature sensor type.

#### **temperature**→**get\_unit()**

Returns the measuring unit for the temperature.

#### **temperature**→**get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

#### **temperature**→**isOnline()**

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

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

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

#### **temperature**→**load(msValidity)**

Preloads the temperature sensor cache with a specified validity duration.

#### **temperature**→**loadCalibrationPoints(rawValues, refValues)**

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

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

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

#### **temperature**→**nextTemperature()**

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

#### **temperature**→**registerTimedReportCallback(callback)**

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

#### **temperature**→**registerValueCallback(callback)**

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

#### **temperature**→**set\_highestValue(newval)**

Changes the recorded maximal value observed.

#### **temperature**→**set\_logFrequency(newval)**

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()

YTemperature

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)=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 temperature sensor (ex:  
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

temperature→get\_advertisedValue()

YTemperature

temperature→advertisedValue()

temperature.get\_advertisedValue()

---

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()****YTemperature****temperature**→**currentRawValue()****temperature.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in Celsius, as a floating point number.

```
def get_currentRawValue( )
```

**Returns :**

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

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

**temperature**→**get\_currentValue()**

**YTemperature**

**temperature**→**currentValue()**

**temperature.get\_currentValue()**

---

Returns the current value of the temperature, in Celsius, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the temperature, in Celsius, as a floating point number

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

---

**temperature**→**get\_errorMessage()****YTemperature****temperature**→**errorMessage()****temperature.get\_errorMessage()**

---

Returns the error message of the latest error with the temperature 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 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()****YTemperature****temperature**→**friendlyName()****temperature.get\_friendlyName()**

---

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 example: `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()**

**YTemperature**

**temperature**→**functionDescriptor()**

**temperature.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`.

---

**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()`

**YTemperature**

`temperature→hardwareId()`

`temperature.get_hardwareId()`

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the temperature sensor (ex: `RELAYLO1-123456.relay1`)

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

---

**temperature**→**get\_highestValue()****YTemperature****temperature**→**highestValue()****temperature.get\_highestValue()**

---

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`.

temperature→get\_logFrequency()

YTemperature

temperature→logFrequency()

temperature.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.

---

**temperature**→**get\_logicalName()****YTemperature****temperature**→**logicalName()****temperature.get\_logicalName()**

---

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 on-line.

**Returns :**

an instance of `YModule`

**temperature**→**get\_recordedData()**

**YTemperature**

**temperature**→**recordedData()**

**temperature.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.

---

`temperature`→`get_reportFrequency()`

`YTemperature`

`temperature`→`reportFrequency()`

`temperature.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`.

**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()****YTemperature****temperature**→**sensorType()****temperature.get\_sensorType()**

---

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()****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()****YTemperature****temperature**→**setLogFrequency()****temperature.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.

**temperature**→**set\_logicalName()**

**YTemperature**

**temperature**→**setLogicalName()**

**temperature.set\_logicalName()**

---

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()****YTemperature****temperature**→**setLowestValue()****temperature.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.

**temperature→set\_reportFrequency()**

**YTemperature**

**temperature→setReportFrequency()**

**temperature.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.

**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 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()****YTemperature****temperature**→**setUserData()****temperature.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

## 3.43. 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');
c++	#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, in degrees, as a floating point number.

#### tilt→get\_currentValue()

Returns the current value of the inclination, in degrees, as a floating point number.

#### 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`.

**tilt→get\_highestValue()**

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

**tilt→get\_logFrequency()**

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

**tilt→get\_logicalName()**

Returns the logical name of the tilt sensor.

**tilt→get\_lowestValue()**

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

**tilt→get\_module()**

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

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

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

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

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

**tilt→get\_reportFrequency()**

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

**tilt→get\_resolution()**

Returns the resolution of the measured values.

**tilt→get\_unit()**

Returns the measuring unit for the inclination.

**tilt→get\_userData()**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

**tilt→isOnline()**

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

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

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

**tilt→load(msValidity)**

Preloads the tilt sensor cache with a specified validity duration.

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

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

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

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

**tilt→nextTilt()**

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

**tilt→registerTimedReportCallback(callback)**

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

**tilt→registerValueCallback(callback)**

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

**tilt→set\_highestValue(newval)**

Changes the recorded maximal value observed.

**tilt→set\_logFrequency(newval)**

Changes the datalogger recording frequency for this function.

**tilt→set\_logicalName(newval)**

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](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.

**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` 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)=RELAYLO1-123456.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 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, in degrees, as a floating point number.

```
def get_currentRawValue( )
```

**Returns :**

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

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, in degrees, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the inclination, in degrees, as a floating point number

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

**tilt**→**get\_errorMessage()**

**YTilt**

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

---

Returns the error message of the latest error with the tilt 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 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

**tilt**→**get\_friendlyName()**

**YTilt**

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

---

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 example: `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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the tilt sensor (ex: `RELAYLO1-123456.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`.

**tilt**→**get\_logicalName()**

**YTilt**

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

---

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()**

**YTilt**

**tilt**→**module()****tilt.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`

**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.

**tilt**→**get\_reportFrequency()**

**YTilt**

**tilt**→**reportFrequency()****tilt.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.

---

**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()**

**YTilt**

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

---

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.

**tilt**→**set\_highestValue()**

**YTilt**

**tilt**→**setHighestValue()****tilt.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.

---

**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.

**tilt**→**set\_logicalName()****YTilt****tilt**→**setLogicalName()****tilt.set\_logicalName()**

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( 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

## 3.44. 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>
nodejs	var yoctolib = require('yoctolib'); var YVoc = yoctolib.YVoc;
php	require_once('yocto_voc.php');
c++	#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 uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.

#### voc→get\_currentValue()

Returns the current value of the estimated VOC concentration, in ppm (vol), as a floating point number.

#### 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 since the device was started.

#### **`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 since the device was started.

#### **`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.

**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.

**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()****YVoc****yFindVoc()YVoc.FindVoc()**

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()**  
**yFirstVoc()YVoc.FirstVoc()**

---

**YVoc**

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` 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 Volatile Organic Compound sensor (ex:  
`Relay(MyCustomName.relay1)=RELAYLO1-123456.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 uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.

```
def get_currentRawValue() ( )
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number

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

**voc**→**get\_currentValue()**

**YVoc**

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

---

Returns the current value of the estimated VOC concentration, in ppm (vol), as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the estimated VOC concentration, in ppm (vol), as a floating point number

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()**

**YVoc**

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

---

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 example: `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()**

**YVoc**

**voc**→**functionDescriptor()**

**voc.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.

---

**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`.

**voc**→**get\_hardwareId()**

**YVoc**

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

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the Volatile Organic Compound sensor (ex: `RELAYLO1-123456.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 since the device was started.

```
def get_highestValue( )
```

**Returns :**

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

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 since the device was started.

```
def get_lowestValue() ( )
```

**Returns :**

a floating point number corresponding to the minimal value observed for the estimated VOC concentration since the device was started

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 on-line.

**Returns :**

an instance of YModule

**voc**→**get\_recordedData()****YVoc****voc**→**recordedData()****voc.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.

---

**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()**

**YVoc**

**voc**→**resolution()****voc.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`.

---

**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`.

**YVoc**  
**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.

```
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.

**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.

```
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.



---

**voc**→**set\_reportFrequency()****YVoc****voc**→**setReportFrequency()****voc.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.

**voc**→**set\_resolution()**

**YVoc**

**voc**→**setResolution()****voc.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.

---

**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

## 3.45. 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');
c++	#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, in Volt, as a floating point number.

#### voltage→get\_currentValue()

Returns the current value of the voltage, in Volt, as a floating point number.

#### 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 since the device was started.

**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 since the device was started.

**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.

**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.

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

Changes the timed value notification frequency for this function.

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

Changes the resolution of the measured physical 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() yFindVoltage()YVoltage.FindVoltage()

YVoltage

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()**

**YVoltage**

**yFirstVoltage()YVoltage.FirstVoltage()**

---

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` 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)=RELAYLO1-123456.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 voltage sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

---

**voltage**→**get\_advertisedValue()**

**YVoltage**

**voltage**→**advertisedValue()**

**voltage.get\_advertisedValue()**

---

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()**

**YVoltage**

**voltage**→**currentRawValue()**

**voltage.get\_currentRawValue()**

---

Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.

```
def get_currentRawValue()
```

**Returns :**

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

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

---

**voltage**→**get\_currentValue()****YVoltage****voltage**→**currentValue()****voltage.get\_currentValue()**

---

Returns the current value of the voltage, in Volt, as a floating point number.

```
def get_currentValue( )
```

**Returns :**

a floating point number corresponding to the current value of the voltage, in Volt, as a floating point number

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

**voltage**→**get\_friendlyName()**

**YVoltage**

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

---

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 example: `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()****YVoltage****voltage**→**functionDescriptor()****voltage.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`.

**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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the voltage sensor (ex: `RELAYLO1-123456.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 since the device was started.

```
def get_highestValue()
```

**Returns :**

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

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 since the device was started.

```
def get_lowestValue( )
```

**Returns :**

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

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 on-line.

**Returns :**

an instance of YModule



---

**voltage**→**get\_recordedData()****YVoltage****voltage**→**recordedData()****voltage.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.

**voltage**→**get\_reportFrequency()**

**YVoltage**

**voltage**→**reportFrequency()**

**voltage.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.

---

**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()**

**YVoltage**

**voltage**→**setHighestValue()**

**voltage.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.

---

**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.

```
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.

**voltage**→**set\_reportFrequency()**

**YVoltage**

**voltage**→**setReportFrequency()**

**voltage.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.



---

**voltage**→**set\_resolution()****YVoltage****voltage**→**setResolution()****voltage.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.

**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( 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. 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	
<b>yFindVSource(func)</b>	Retrieves a voltage source for a given identifier.
<b>yFirstVSource()</b>	Starts the enumeration of voltage sources currently accessible.
YVSource methods	
<b>vsource→describe()</b>	Returns a short text that describes the function in the form TYPE ( NAME ) =SERIAL . FUNCTIONID.
<b>vsource→get_advertisedValue()</b>	Returns the current value of the voltage source (no more than 6 characters).
<b>vsource→get_errorMessage()</b>	Returns the error message of the latest error with this function.
<b>vsource→get_errorType()</b>	Returns the numerical error code of the latest error with this function.
<b>vsource→get_extPowerFailure()</b>	Returns true if external power supply voltage is too low.
<b>vsource→get_failure()</b>	Returns true if the module is in failure mode.
<b>vsource→get_friendlyName()</b>	Returns a global identifier of the function in the format MODULE_NAME . FUNCTION_NAME.
<b>vsource→get_functionDescriptor()</b>	Returns a unique identifier of type YFUN_DESCR corresponding to the function.
<b>vsource→get_functionId()</b>	Returns the hardware identifier of the function, without reference to the module.
<b>vsource→get_hardwareId()</b>	Returns the unique hardware identifier of the function in the form SERIAL . FUNCTIONID.
<b>vsource→get_logicalName()</b>	Returns the logical name of the voltage source.
<b>vsource→get_module()</b>	Gets the YModule object for the device on which the function is located.
<b>vsource→get_module_async(callback, context)</b>	

### 3. Reference

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

#### **`vsource`**→**`get_overCurrent()`**

Returns true if the appliance connected to the device is too greedy .

#### **`vsource`**→**`get_overHeat()`**

Returns TRUE if the module is overheating.

#### **`vsource`**→**`get_overLoad()`**

Returns true if the device is not able to maintain the requested voltage output .

#### **`vsource`**→**`get_regulationFailure()`**

Returns true if the voltage output is too high regarding the requested voltage .

#### **`vsource`**→**`get_unit()`**

Returns the measuring unit for the voltage.

#### **`vsource`**→**`get_userData()`**

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

#### **`vsource`**→**`get_voltage()`**

Returns the voltage output command (mV)

#### **`vsource`**→**`isOnline()`**

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

#### **`vsource`**→**`isOnline_async(callback, context)`**

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

#### **`vsource`**→**`load(msValidity)`**

Preloads the function cache with a specified validity duration.

#### **`vsource`**→**`load_async(msValidity, callback, context)`**

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

#### **`vsource`**→**`nextVSource()`**

Continues the enumeration of voltage sources started using `yFirstVSource()` .

#### **`vsource`**→**`pulse(voltage, ms_duration)`**

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

#### **`vsource`**→**`registerValueCallback(callback)`**

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

#### **`vsource`**→**`set_logicalName(newval)`**

Changes the logical name of the voltage source.

#### **`vsource`**→**`set_userData(data)`**

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

#### **`vsource`**→**`set_voltage(newval)`**

Tunes the device output voltage (milliVolts).

#### **`vsource`**→**`voltageMove(target, ms_duration)`**

Performs a smooth move at constant speed toward a given value.

#### **`vsource`**→**`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()****YVSource****vsource**→**advertisedValue()****vsource.get\_advertisedValue()**


---

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).

js	function <b>get_advertisedValue</b> ( )
php	function <b>get_advertisedValue</b> ( )
cpp	string <b>get_advertisedValue</b> ( )
m	-(NSString*) advertisedValue
pas	function <b>get_advertisedValue</b> ( ): string
vb	function <b>get_advertisedValue</b> ( ) As String
cs	string <b>get_advertisedValue</b> ( )
java	String <b>get_advertisedValue</b> ( )
py	def <b>get_advertisedValue</b> ( )
cmd	YVSource <b>target</b> <b>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.

**vsource**→**get\_errorMessage()**

**YVSource**

**vsource**→**errorMessage()**

**vsource.get\_errorMessage()**

---

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 occurred 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

---

**vsource**→**get\_extPowerFailure()**

YVSource

**vsource**→**extPowerFailure()****vsource.get\_extPowerFailure()**

---

Returns true if external power supply voltage is too low.

```
def get_extPowerFailure( )
```

**vsource**→**get\_extPowerFailure()****vsource**→**extPowerFailure()****vsource.get\_extPowerFailure()**

---

Returns true if external power supply voltage is too low.

js	function <b>get_extPowerFailure</b> ( )
php	function <b>get_extPowerFailure</b> ( )
cpp	Y_EXTPOWERFAILURE_enum <b>get_extPowerFailure</b> ( )
m	-(Y_EXTPOWERFAILURE_enum) extPowerFailure
pas	function <b>get_extPowerFailure</b> ( ): Integer
vb	function <b>get_extPowerFailure</b> ( ) As Integer
cs	int <b>get_extPowerFailure</b> ( )
java	int <b>get_extPowerFailure</b> ( )
py	def <b>get_extPowerFailure</b> ( )
cmd	YVSource <b>target</b> <b>get_extPowerFailure</b>

**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()****YVSource****vsource**→**failure()****vsource.get\_failure()**

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 <b>get_failure()</b>
php	function <b>get_failure()</b>
cpp	Y_FAILURE_enum <b>get_failure()</b>
m	-(Y_FAILURE_enum) failure
pas	function <b>get_failure()</b> : Integer
vb	function <b>get_failure()</b> As Integer
cs	int <b>get_failure()</b>
java	int <b>get_failure()</b>
py	def <b>get_failure()</b>
cmd	YVSource <b>target get_failure</b>

More information can be obtained by testing `get_overheat`, `get_overcurrent` etc... When a error condition is met, the output voltage is set to zéro 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`.

**vsource**→**get\_functionDescriptor()**

**YVSource**

**vsource**→**functionDescriptor()**

**vsource.get\_vsourceDescriptor()**

---

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 <b>get_logicalName</b> ( )
php	function <b>get_logicalName</b> ( )
cpp	string <b>get_logicalName</b> ( )
m	-(NSString*) logicalName
pas	function <b>get_logicalName</b> ( ): string
vb	function <b>get_logicalName</b> ( ) As String
cs	string <b>get_logicalName</b> ( )
java	String <b>get_logicalName</b> ( )
py	def <b>get_logicalName</b> ( )
cmd	YVSource <b>target</b> <b>get_logicalName</b>

**Returns :**

a string corresponding to the logical name of the voltage source

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

**vsource**→**get\_module()**

**YVSource**

**vsource**→**module()****vsource.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

**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 <b>get_overCurrent</b> ( )
php	function <b>get_overCurrent</b> ( )
cpp	Y_OVERCURRENT_enum <b>get_overCurrent</b> ( )
m	-(Y_OVERCURRENT_enum) overCurrent
pas	function <b>get_overCurrent</b> ( ): Integer
vb	function <b>get_overCurrent</b> ( ) As Integer
cs	int <b>get_overCurrent</b> ( )
java	int <b>get_overCurrent</b> ( )
py	def <b>get_overCurrent</b> ( )
cmd	YVSource <b>target</b> <b>get_overCurrent</b>

**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.

**vsource**→**get\_overHeat()****vsource**→**overHeat()****vsource.get\_overHeat()**

---

Returns TRUE if the module is overheating.

```
def get_overHeat( )
```

**vsource**→**get\_overHeat()****vsource**→**overHeat()****vsource.get\_overHeat()**

---

Returns TRUE if the module is overheating.

js	function <b>get_overHeat</b> ( )
php	function <b>get_overHeat</b> ( )
cpp	Y_OVERHEAT_enum <b>get_overHeat</b> ( )
m	-(Y_OVERHEAT_enum) overHeat
pas	function <b>get_overHeat</b> ( ): Integer
vb	function <b>get_overHeat</b> ( ) As Integer
cs	int <b>get_overHeat</b> ( )
java	int <b>get_overHeat</b> ( )
py	def <b>get_overHeat</b> ( )
cmd	YVSource <b>target</b> <b>get_overHeat</b>

**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 maintaint the requested voltage output .

```
def get_overLoad( )
```

**vsource**→**get\_overLoad()****vsource**→**overLoad()****vsource.get\_overLoad()**


---

Returns true if the device is not able to maintaint the requested voltage output .

js	function <b>get_overLoad</b> ( )
php	function <b>get_overLoad</b> ( )
cpp	Y_OVERLOAD_enum <b>get_overLoad</b> ( )
m	-(Y_OVERLOAD_enum) overLoad
pas	function <b>get_overLoad</b> ( ): Integer
vb	function <b>get_overLoad</b> ( ) As Integer
cs	int <b>get_overLoad</b> ( )
java	int <b>get_overLoad</b> ( )
py	def <b>get_overLoad</b> ( )
cmd	YVSource <b>target</b> <b>get_overLoad</b>

**Returns :**

either Y\_OVERLOAD\_FALSE or Y\_OVERLOAD\_TRUE, according to true if the device is not able to maintaint the requested voltage output

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

**vsource**→**get\_regulationFailure()****YVSource****vsource**→**regulationFailure()****vsource.get\_regulationFailure()**

Returns true if the voltage output is too high regarding the requested voltage .

```
def get_regulationFailure( )
```

**vsource**→**get\_regulationFailure()****vsource**→**regulationFailure(vsource.get\_regulationFailure())**

Returns true if the voltage output is too high regarding the requested voltage .

js	function <b>get_regulationFailure</b> ( )
php	function <b>get_regulationFailure</b> ( )
cpp	Y_REGULATIONFAILURE_enum <b>get_regulationFailure</b> ( )
m	-(Y_REGULATIONFAILURE_enum) regulationFailure
pas	function <b>get_regulationFailure</b> ( ): Integer
vb	function <b>get_regulationFailure</b> ( ) As Integer
cs	int <b>get_regulationFailure</b> ( )
java	int <b>get_regulationFailure</b> ( )
py	def <b>get_regulationFailure</b> ( )
cmd	YVSource <b>target</b> <b>get_regulationFailure</b>

**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 <b>get_unit</b> ( )
php	function <b>get_unit</b> ( )
cpp	string <b>get_unit</b> ( )
m	-(NSString*) unit
pas	function <b>get_unit</b> ( ): string
vb	function <b>get_unit</b> ( ) As String
cs	string <b>get_unit</b> ( )
java	String <b>get_unit</b> ( )
py	def <b>get_unit</b> ( )
cmd	YVSource <b>target</b> <b>get_unit</b>

**Returns :**

a string corresponding to the measuring unit for the voltage

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

**vsource**→**get\_userData()**

**YVSource**

**vsource**→**userData()****vsource.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()****vsource.pulse()****YVSource**

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

```
def pulse( voltage, ms_duration)
```

**vsource**→**pulse()****vsource.pulse()**

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

js	function <b>pulse</b> ( <b>voltage</b> , <b>ms_duration</b> )
php	function <b>pulse</b> ( <b>\$voltage</b> , <b>\$ms_duration</b> )
cpp	int <b>pulse</b> ( int <b>voltage</b> , int <b>ms_duration</b> )
m	-(int) <b>pulse</b> : (int) <b>voltage</b> : (int) <b>ms_duration</b>
pas	function <b>pulse</b> ( <b>voltage</b> : integer, <b>ms_duration</b> : integer): integer
vb	function <b>pulse</b> ( ByVal <b>voltage</b> As Integer, ByVal <b>ms_duration</b> As Integer) As Integer
cs	int <b>pulse</b> ( int <b>voltage</b> , int <b>ms_duration</b> )
java	int <b>pulse</b> ( int <b>voltage</b> , int <b>ms_duration</b> )
py	def <b>pulse</b> ( <b>voltage</b> , <b>ms_duration</b> )
cmd	YVSource <b>target pulse voltage ms_duration</b>

**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.

**vsource**→**registerValueCallback()**  
**vsource.registerValueCallback()**

YVSource

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

```
def registerValueCallback( callback )
```

**vsource**→**registerValueCallback()****vsource.registerValueCallback()**

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

js	function <b>registerValueCallback</b> ( <b>callback</b> )
php	function <b>registerValueCallback</b> ( <b>\$callback</b> )
cpp	void <b>registerValueCallback</b> ( YDisplayUpdateCallback <b>callback</b> )
pas	procedure <b>registerValueCallback</b> ( <b>callback</b> : TGenericUpdateCallback )
vb	procedure <b>registerValueCallback</b> ( ByVal <b>callback</b> As GenericUpdateCallback )
cs	void <b>registerValueCallback</b> ( UpdateCallback <b>callback</b> )
java	void <b>registerValueCallback</b> ( UpdateCallback <b>callback</b> )
py	def <b>registerValueCallback</b> ( <b>callback</b> )
m	-(void) <b>registerValueCallback</b> : (YFunctionUpdateCallback) <b>callback</b>

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()****YVSource****vsource**→**setLogicalName()****vsource.set\_logicalName()**


---

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</b> ( newval)
php	function <b>set_logicalName</b> ( \$newval)
cpp	int <b>set_logicalName</b> ( const string& newval)
m	-(int) setLogicalName : (NSString*) newval
pas	function <b>set_logicalName</b> ( newval: string): integer
vb	function <b>set_logicalName</b> ( ByVal newval As String) As Integer
cs	int <b>set_logicalName</b> ( string newval)
java	int <b>set_logicalName</b> ( String newval)
py	def <b>set_logicalName</b> ( newval)
cmd	YVSource <b>target set_logicalName</b> 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.

**vsource**→**set\_userdata()**

**YVSource**

**vsource**→**setUserData()****vsource.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

**vsource**→**set\_voltage()****YVSource****vsource**→**setVoltage()****vsource.set\_voltage()**

Tunes the device output voltage (milliVolts).

def **set\_voltage**( **newval**)**vsource**→**set\_voltage()****vsource**→**setVoltage()****vsource.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.

## 3.47. 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>
nodejs	var yoctolib = require('yoctolib'); var YWakeUpMonitor = yoctolib.YWakeUpMonitor;
php	require_once('yocto_wakeupmonitor.php');
c++	#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()

### 3. Reference

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(data)</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)=RELAYLO1-123456.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 monitor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

---

**wakeupmonitor**→**get\_advertisedValue()****YWakeUpMonitor****wakeupmonitor**→**advertisedValue()****wakeupmonitor.get\_advertisedValue()**

---

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()

YWakeUpMonitor

wakeupmonitor→errorMessage()

wakeupmonitor.get\_errorMessage()

---

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()

YWakeUpMonitor

wakeupmonitor→friendlyName()

wakeupmonitor.get\_friendlyName()

---

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 example: `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()****YWakeUpMonitor****wakeupmonitor**→**functionDescriptor()****wakeupmonitor.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`.

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()****YWakeUpMonitor****wakeupmonitor**→**hardwareId()****wakeupmonitor.get\_hardwareId()**

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the monitor (ex: `RELAYLO1-123456.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()****YWakeUpMonitor****wakeupmonitor**→**module()****wakeupmonitor.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

wakeupmonitor→get\_nextWakeUp()

YWakeUpMonitor

wakeupmonitor→nextWakeUp()

wakeupmonitor.get\_nextWakeUp()

---

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()

YWakeUpMonitor

wakeupmonitor→sleepCountdown()

wakeupmonitor.get\_sleepCountdown()

---

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\_SLEEPDOWNDOWN\_INVALID.



---

**wakeupmonitor**→**get\_userData()****YWakeUpMonitor****wakeupmonitor**→**userData()****wakeupmonitor.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.

wakeupmonitor→get\_wakeUpReason()

YWakeUpMonitor

wakeupmonitor→wakeUpReason()

wakeupmonitor.get\_wakeUpReason()

---

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\_SCHEDULE1 and Y\_WAKEUPREASON\_SCHEDULE2 corresponding to the latest wake up reason

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

---

**wakeupmonitor**→**get\_wakeUpState()****YWakeUpMonitor****wakeupmonitor**→**wakeUpState()****wakeupmonitor.get\_wakeUpState()**

---

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()****YWakeUpMonitor****wakeupmonitor**→**setLogicalName()****wakeupmonitor.set\_logicalName()**

---

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()

YWakeUpMonitor

wakeupmonitor→setNextWakeUp()

wakeupmonitor.set\_nextWakeUp()

---

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()****YWakeUpMonitor****wakeupmonitor**→**setPowerDuration()****wakeupmonitor.set\_powerDuration()**

---

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()****YWakeUpMonitor****wakeupmonitor**→**setUserData()****wakeupmonitor.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

**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** number of seconds before next wake up

**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( )
```

## 3.48. 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;
php	require_once('yocto_wakeupschedule.php');
c++	#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.

**wakeupschedule→get\_minutesB()**

Returns the minutes in the 30-59 interval of each hour scheduled for wake up.

**wakeupschedule→get\_module()**

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

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

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

**wakeupschedule→get\_monthDays()**

Returns the days of the month scheduled for wake up.

**wakeupschedule→get\_months()**

Returns the months scheduled for wake up.

**wakeupschedule→get\_nextOccurence()**

Returns the date/time (seconds) of the next wake up occurrence

**wakeupschedule→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set\_userData.

**wakeupschedule→get\_weekDays()**

Returns the days of the week scheduled for wake up.

**wakeupschedule→isOnline()**

Checks if the wake up schedule is currently reachable, without raising any error.

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

Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).

**wakeupschedule→load(msValidity)**

Preloads the wake up schedule cache with a specified validity duration.

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

Preloads the wake up schedule cache with a specified validity duration (asynchronous version).

**wakeupschedule→nextWakeUpSchedule()**

Continues the enumeration of wake up schedules started using yFirstWakeUpSchedule().

**wakeupschedule→registerValueCallback(callback)**

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

**wakeupschedule→set\_hours(newval)**

Changes the hours when a wake up must take place.

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

Changes the logical name of the wake up schedule.

**wakeupschedule→set\_minutes(bitmap)**

Changes all the minutes where a wake up must take place.

**wakeupschedule→set\_minutesA(newval)**

Changes the minutes in the 00-29 interval when a wake up must take place.

**wakeupschedule→set\_minutesB(newval)**

Changes the minutes in the 30-59 interval when a wake up must take place.

**wakeupschedule→set\_monthDays(newval)**

Changes the days of the month when a wake up must take place.

**wakeupschedule→set\_months(newval)**

Changes the months when a wake up must take place.

**wakeupschedule→set\_userData(data)**

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()  
yFirstWakeUpSchedule()  
YWakeUpSchedule.FirstWakeUpSchedule()**

---

**YWakeUpSchedule**

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` 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 wake up schedule (ex:  
`Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

`wakeupschedule`→`get_advertisedValue()`

`YWakeUpSchedule`

`wakeupschedule`→`advertisedValue()`

`wakeupschedule.get_advertisedValue()`

---

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→get_errorMessage()`

**YWakeUpSchedule**

`wakeupschedule→errorMessage()`

`wakeupschedule.get_errorMessage()`

---

Returns the error message of the latest error with the wake up schedule.

```
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 wake up schedule object

`wakeupschedule`→`get_errorType()`

**YWakeUpSchedule**

`wakeupschedule`→`errorType()`

`wakeupschedule.get_errorType()`

---

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()****YWakeUpSchedule****wakeupschedule→friendlyName()****wakeupschedule.get\_friendlyName()**

---

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 example: `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()`

`YWakeUpSchedule`

`wakeupschedule`→`functionDescriptor()`

`wakeupschedule.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`.

---

**wakeupschedule→get\_functionId()****YWakeUpSchedule****wakeupschedule→functionId()****wakeupschedule.get\_functionId()**

---

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()`

`YWakeUpSchedule`

`wakeupschedule`→`hardwareId()`

`wakeupschedule.get_hardwareId()`

---

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 `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the wake up schedule (ex: `RELAYLO1-123456.relay1`)

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

---

**wakeupschedule→get\_hours()**

**YWakeUpSchedule**

**wakeupschedule→hours()**

**wakeupschedule.get\_hours()**

---

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()**

**YWakeUpSchedule**

**wakeupschedule**→**logicalName()**

**wakeupschedule.get\_logicalName()**

---

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()**

**YWakeUpSchedule**

**wakeupschedule→minutes()**

**wakeupschedule.get\_minutes()**

---

Returns all the minutes of each hour that are scheduled for wake up.

```
def get_minutes( )
```

**wakeupschedule**→**get\_minutesA()**

**YWakeUpSchedule**

**wakeupschedule**→**minutesA()**

**wakeupschedule.get\_minutesA()**

---

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()**

**YWakeUpSchedule**

**wakeupschedule→minutesB()**

**wakeupschedule.get\_minutesB()**

---

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()**

**YWakeUpSchedule**

**wakeupschedule**→**module()**

**wakeupschedule.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

---

**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()****YWakeUpSchedule****wakeupschedule**→**nextOccurence()****wakeupschedule.get\_nextOccurence()**

---

Returns the date/time (seconds) of the next wake up occurrence

```
def get_nextOccurence( )
```

**Returns :**

an integer corresponding to the date/time (seconds) of the next wake up occurrence

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()**

**YWakeUpSchedule**

**wakeupschedule→weekDays()**

**wakeupschedule.get\_weekDays()**

---

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()`

**YWakeUpSchedule**

`wakeupschedule`→`setHours()`

`wakeupschedule.set_hours()`

---

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()****YWakeUpSchedule****wakeupschedule**→**setLogicalName()****wakeupschedule.set\_logicalName()**

---

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()**

**YWakeUpSchedule**

**wakeupschedule→setMinutes()**

**wakeupschedule.set\_minutes()**

---

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()****YWakeUpSchedule****wakeupschedule→setMinutesA()****wakeupschedule.set\_minutesA()**

---

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()

YWakeUpSchedule

wakeupschedule→setMinutesB()

wakeupschedule.set\_minutesB()

---

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()****YWakeUpSchedule****wakeupschedule**→**setMonthDays()****wakeupschedule.set\_monthDays()**

---

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()`

`YWakeUpSchedule`

`wakeupschedule`→`setMonths()`

`wakeupschedule.set_months()`

---

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()****YWakeUpSchedule****wakeupschedule**→**setUserData()****wakeupschedule.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

`wakeupschedule`→`set_weekDays()`

`YWakeUpSchedule`

`wakeupschedule`→`setWeekDays()`

`wakeupschedule.set_weekDays()`

---

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.

## 3.49. 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>
nodejs	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.

**watchdog→get\_hardwareId()**

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

**watchdog→get\_logicalName()**

Returns the logical name of the watchdog.

**watchdog→get\_maxTimeOnStateA()**

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state A before automatically switching back in to B state.

**watchdog→get\_maxTimeOnStateB()**

Retourne the maximum time (ms) allowed for \$THEFUNCTIONS\$ to stay in state B before automatically switching back in to A state.

**watchdog→get\_module()**

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

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

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

**watchdog→get\_output()**

Returns the output state of the watchdog, when used as a simple switch (single throw).

**watchdog→get\_pulseTimer()**

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

**watchdog→get\_running()**

Returns the watchdog running state.

**watchdog→get\_state()**

Returns the state of the watchdog (A for the idle position, B for the active position).

**watchdog→get\_stateAtPowerOn()**

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

**watchdog→get\_triggerDelay()**

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

**watchdog→get\_triggerDuration()**

Returns the duration of resets caused by the watchdog, in milliseconds.

**watchdog→get\_userData()**

Returns the value of the userData attribute, as previously stored using method set\_userData.

**watchdog→isOnline()**

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

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

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

**watchdog→load(msValidity)**

Preloads the watchdog cache with a specified validity duration.

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

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

**watchdog→nextWatchdog()**

Continues the enumeration of watchdog started using yFirstWatchdog( ).

**watchdog→pulse(ms\_duration)**

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

**watchdog→registerValueCallback(callback)**

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()

YWatchdog

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()  
yFirstWatchdog()YWatchdog.FirstWatchdog()**

---

**YWatchdog**

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 milliseconds

**ms\_duration** pulse duration, in milliseconds

**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)=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 watchdog (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

**watchdog**→**get\_advertisedValue()**

**YWatchdog**

**watchdog**→**advertisedValue()**

**watchdog.get\_advertisedValue()**

---

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 running state at module power on.

```
def get_autoStart( )
```

**Returns :**

either Y\_AUTOSTART\_OFF or Y\_AUTOSTART\_ON, according to the watchdog running 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()****YWatchdog****watchdog→errorMessage()****watchdog.get\_errorMessage()**

---

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()****YWatchdog****watchdog**→**friendlyName()****watchdog.get\_friendlyName()**

---

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 example: `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()**

**YWatchdog**

**watchdog**→**functionDescriptor()**

**watchdog.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`.

---

**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()****YWatchdog****watchdog**→**logicalName()****watchdog.get\_logicalName()**

---

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()**

**YWatchdog**

**watchdog**→**maxTimeOnStateA()**

**watchdog.get\_maxTimeOnStateA()**

---

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()****YWatchdog****watchdog**→**maxTimeOnStateB()****watchdog.get\_maxTimeOnStateB()**

---

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 on-line.

**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_PULSETIMER_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()****YWatchdog****watchdog→stateAtPowerOn()****watchdog.get\_stateAtPowerOn()**

---

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()**

**YWatchdog**

**watchdog**→**triggerDelay()**

**watchdog.get\_triggerDelay()**

---

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()****YWatchdog****watchdog→triggerDuration()****watchdog.get\_triggerDuration()**

---

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 milliseconds

### 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()**

**YWatchdog**

**watchdog**→**setLogicalName()**

**watchdog.set\_logicalName()**

---

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()****YWatchdog****watchdog**→**setMaxTimeOnStateA()****watchdog.set\_maxTimeOnStateA()**

---

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()**

**YWatchdog**

**watchdog**→**setMaxTimeOnStateB()**

**watchdog.set\_maxTimeOnStateB()**

---

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()**

**YWatchdog**

**watchdog**→**setStateAtPowerOn()**

**watchdog.set\_stateAtPowerOn()**

---

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

## 3.50. 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');
c++	#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→softAPNetwork(ssid, securityKey)**

Changes the configuration of the wireless lan interface to create a new wireless network by emulating a WiFi access point (Soft AP).

**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()  
yFirstWireless()YWireless.FirstWireless()**

---

**YWireless**

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)
```

On the YoctoHub-Wireless-g, it is best to use softAPNetworkInstead(), which emulates an access point (Soft AP) which is more efficient and more widely supported than ad-hoc networks.

When a security key is specified for an ad-hoc network, the network is protected by a WEP40 key (5 characters or 10 hexadecimal digits) or WEP128 key (13 characters or 26 hexadecimal digits). It is recommended to use a well-randomized WEP128 key using 26 hexadecimal digits to maximize security. 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 when 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)=RELAYLO1-123456.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)=RELAYLO1-123456.relay1`)

**wireless**→**get\_advertisedValue()**

**YWireless**

**wireless**→**advertisedValue()**

**wireless.get\_advertisedValue()**

---

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()**

**YWireless**

**wireless**→**detectedWlans()**

**wireless.get\_detectedWlans()**

---

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 access 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()****YWireless****wireless**→**errorMessage()****wireless.get\_errorMessage()**

---

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 example: `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\_hardwareId()**

**YWireless**

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

---

Returns the unique hardware identifier of the wireless lan 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 wireless lan interface (for example `RELAYLO1-123456.relay1`).

**Returns :**

a string that uniquely identifies the wireless lan interface (ex: `RELAYLO1-123456.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 when 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( 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



**wireless**→**softAPNetwork()****wireless.softAPNetwork()****YWireless**

Changes the configuration of the wireless lan interface to create a new wireless network by emulating a WiFi access point (Soft AP).

```
def softAPNetwork( ssid, securityKey)
```

This function can only be used with the YoctoHub-Wireless-g.

When a security key is specified for a SoftAP network, the network is protected by a WEP40 key (5 characters or 10 hexadecimal digits) or WEP128 key (13 characters or 26 hexadecimal digits). It is recommended to use a well-randomized WEP128 key using 26 hexadecimal digits to maximize security. 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 when the call succeeds.

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



# Index

## A

Accelerometer 32  
adhocNetwork, YWireless 1763  
Altitude 74  
AnButton 116

## B

Blueprint 10  
brakingForceMove, YMotor 880

## C

calibrate, YLightSensor 743  
calibrateFromPoints, YAccelerometer 36  
calibrateFromPoints, YAltitude 78  
calibrateFromPoints, YCarbonDioxide 158  
calibrateFromPoints, YCompass 226  
calibrateFromPoints, YCurrent 266  
calibrateFromPoints, YGenericSensor 552  
calibrateFromPoints, YGyro 601  
calibrateFromPoints, YHumidity 677  
calibrateFromPoints, YLightSensor 744  
calibrateFromPoints, YMagnetometer 785  
calibrateFromPoints, YPower 1001  
calibrateFromPoints, YPressure 1044  
calibrateFromPoints, YPwmInput 1083  
calibrateFromPoints, YQt 1192  
calibrateFromPoints, YSensor 1330  
calibrateFromPoints, YTemperature 1461  
calibrateFromPoints, YTilt 1502  
calibrateFromPoints, YVoc 1541  
calibrateFromPoints, YVoltage 1580  
callbackLogin, YNetwork 922  
cancel3DCalibration, YRefFrame 1258  
CarbonDioxide 154  
checkFirmware, YModule 833  
CheckLogicalName, YAPI 12  
clear, YDisplayLayer 464  
clearConsole, YDisplayLayer 465  
Clock 1227  
ColorLed 193  
Compass 222  
Configuration 1254  
consoleOut, YDisplayLayer 466  
copyLayerContent, YDisplay 420  
Current 262

## D

Data 335, 345, 357  
DataLogger 301  
delayedPulse, YDigitalIO 376  
delayedPulse, YRelay 1294  
delayedPulse, YWatchdog 1719

describe, YAccelerometer 37  
describe, YAltitude 79  
describe, YAnButton 120  
describe, YCarbonDioxide 159  
describe, YColorLed 196  
describe, YCompass 227  
describe, YCurrent 267  
describe, YDataLogger 305  
describe, YDigitalIO 377  
describe, YDisplay 421  
describe, YDualPower 498  
describe, YFiles 523  
describe, YGenericSensor 553  
describe, YGyro 602  
describe, YHubPort 651  
describe, YHumidity 678  
describe, YLed 715  
describe, YLightSensor 745  
describe, YMagnetometer 786  
describe, YModule 834  
describe, YMotor 881  
describe, YNetwork 923  
describe, YOsControl 977  
describe, YPower 1002  
describe, YPressure 1045  
describe, YPwmInput 1084  
describe, YPwmOutput 1131  
describe, YPwmPowerSource 1168  
describe, YQt 1193  
describe, YRealTimeClock 1230  
describe, YRefFrame 1259  
describe, YRelay 1295  
describe, YSensor 1331  
describe, YSerialPort 1370  
describe, YServo 1426  
describe, YTemperature 1462  
describe, YTilt 1503  
describe, YVoc 1542  
describe, YVoltage 1581  
describe, YWakeUpMonitor 1647  
describe, YWakeUpSchedule 1682  
describe, YWatchdog 1720  
describe, YWireless 1764  
Digital 372  
DisableExceptions, YAPI 13  
Display 416  
DisplayLayer 463  
download, YFiles 524  
download, YModule 835  
drawBar, YDisplayLayer 467  
drawBitmap, YDisplayLayer 468  
drawCircle, YDisplayLayer 469  
drawDisc, YDisplayLayer 470  
drawImage, YDisplayLayer 471  
drawPixel, YDisplayLayer 472

drawRect, YDisplayLayer 473  
drawText, YDisplayLayer 474  
drivingForceMove, YMotor 882  
dutyCycleMove, YPwmOutput 1132  
Dynamic 3

## E

EnableExceptions, YAPI 14  
Error 7  
External 495

## F

fade, YDisplay 422  
Files 3, 520  
FindAccelerometer, YAccelerometer 34  
FindAltitude, YAltitude 76  
FindAnButton, YAnButton 118  
FindCarbonDioxide, YCarbonDioxide 156  
FindColorLed, YColorLed 194  
FindCompass, YCompass 224  
FindCurrent, YCurrent 264  
FindDataLogger, YDataLogger 303  
FindDigitalIO, YDigitalIO 374  
FindDisplay, YDisplay 418  
FindDualPower, YDualPower 496  
FindFiles, YFiles 521  
FindGenericSensor, YGenericSensor 550  
FindGyro, YGyro 599  
FindHubPort, YHubPort 649  
FindHumidity, YHumidity 675  
FindLed, YLed 713  
FindLightSensor, YLightSensor 741  
FindMagnetometer, YMagnetometer 783  
FindModule, YModule 831  
FindMotor, YMotor 878  
FindNetwork, YNetwork 920  
FindOsControl, YOsControl 975  
FindPower, YPower 999  
FindPressure, YPressure 1042  
FindPwmInput, YPwmInput 1081  
FindPwmOutput, YPwmOutput 1129  
FindPwmPowerSource, YPwmPowerSource 1166  
FindQt, YQt 1190  
FindRealTimeClock, YRealTimeClock 1228  
FindRefFrame, YRefFrame 1256  
FindRelay, YRelay 1292  
FindSensor, YSensor 1328  
FindSerialPort, YSerialPort 1368  
FindServo, YServo 1424  
FindTemperature, YTemperature 1459  
FindTilt, YTilt 1500  
FindVoc, YVoc 1539  
FindVoltage, YVoltage 1578  
FindVSource, YVSource 1616  
FindWakeUpMonitor, YWakeUpMonitor 1645  
FindWakeUpSchedule, YWakeUpSchedule 1680  
FindWatchdog, YWatchdog 1717

FindWireless, YWireless 1761  
FirstAccelerometer, YAccelerometer 35  
FirstAltitude, YAltitude 77  
FirstAnButton, YAnButton 119  
FirstCarbonDioxide, YCarbonDioxide 157  
FirstColorLed, YColorLed 195  
FirstCompass, YCompass 225  
FirstCurrent, YCurrent 265  
FirstDataLogger, YDataLogger 304  
FirstDigitalIO, YDigitalIO 375  
FirstDisplay, YDisplay 419  
FirstDualPower, YDualPower 497  
FirstFiles, YFiles 522  
FirstGenericSensor, YGenericSensor 551  
FirstGyro, YGyro 600  
FirstHubPort, YHubPort 650  
FirstHumidity, YHumidity 676  
FirstLed, YLed 714  
FirstLightSensor, YLightSensor 742  
FirstMagnetometer, YMagnetometer 784  
FirstModule, YModule 832  
FirstMotor, YMotor 879  
FirstNetwork, YNetwork 921  
FirstOsControl, YOsControl 976  
FirstPower, YPower 1000  
FirstPressure, YPressure 1043  
FirstPwmInput, YPwmInput 1082  
FirstPwmOutput, YPwmOutput 1130  
FirstPwmPowerSource, YPwmPowerSource 1167  
FirstQt, YQt 1191  
FirstRealTimeClock, YRealTimeClock 1229  
FirstRefFrame, YRefFrame 1257  
FirstRelay, YRelay 1293  
FirstSensor, YSensor 1329  
FirstSerialPort, YSerialPort 1369  
FirstServo, YServo 1425  
FirstTemperature, YTemperature 1460  
FirstTilt, YTilt 1501  
FirstVoc, YVoc 1540  
FirstVoltage, YVoltage 1579  
FirstVSource, YVSource 1617  
FirstWakeUpMonitor, YWakeUpMonitor 1646  
FirstWakeUpSchedule, YWakeUpSchedule 1681  
FirstWatchdog, YWatchdog 1718  
FirstWireless, YWireless 1762  
forgetAllDataStreams, YDataLogger 306  
format\_fs, YFiles 525  
Formatted 335  
Frame 1254  
FreeAPI, YAPI 15  
functionCount, YModule 836  
functionId, YModule 837  
functionName, YModule 838  
Functions 11  
functionValue, YModule 839

## G

General 11

GenericSensor 548

get\_3DCalibrationHint, YRefFrame 1260

get\_3DCalibrationLogMsg, YRefFrame 1261

get\_3DCalibrationProgress, YRefFrame 1262

get\_3DCalibrationStage, YRefFrame 1263

get\_3DCalibrationStageProgress, YRefFrame 1264

get\_adminPassword, YNetwork 924

get\_advertisedValue, YAccelerometer 38

get\_advertisedValue, YAltitude 80

get\_advertisedValue, YAnButton 121

get\_advertisedValue, YCarbonDioxide 160

get\_advertisedValue, YColorLed 197

get\_advertisedValue, YCompass 228

get\_advertisedValue, YCurrent 268

get\_advertisedValue, YDataLogger 307

get\_advertisedValue, YDigitalIO 378

get\_advertisedValue, YDisplay 423

get\_advertisedValue, YDualPower 499

get\_advertisedValue, YFiles 526

get\_advertisedValue, YGenericSensor 554

get\_advertisedValue, YGyro 603

get\_advertisedValue, YHubPort 652

get\_advertisedValue, YHumidity 679

get\_advertisedValue, YLed 716

get\_advertisedValue, YLightSensor 746

get\_advertisedValue, YMagnetometer 787

get\_advertisedValue, YMotor 883

get\_advertisedValue, YNetwork 925

get\_advertisedValue, YOsControl 978

get\_advertisedValue, YPower 1003

get\_advertisedValue, YPressure 1046

get\_advertisedValue, YPwmInput 1085

get\_advertisedValue, YPwmOutput 1133

get\_advertisedValue, YPwmPowerSource 1169

get\_advertisedValue, YQt 1194

get\_advertisedValue, YRealTimeClock 1231

get\_advertisedValue, YRefFrame 1265

get\_advertisedValue, YRelay 1296

get\_advertisedValue, YSensor 1332

get\_advertisedValue, YSerialPort 1372

get\_advertisedValue, YServo 1427

get\_advertisedValue, YTemperature 1463

get\_advertisedValue, YTilt 1504

get\_advertisedValue, YVoc 1543

get\_advertisedValue, YVoltage 1582

get\_advertisedValue, YVSource 1618

get\_advertisedValue, YWakeUpMonitor 1648

get\_advertisedValue, YWakeUpSchedule 1683

get\_advertisedValue, YWatchdog 1721

get\_advertisedValue, YWireless 1765

get\_allSettings, YModule 840

get\_analogCalibration, YAnButton 122

get\_autoStart, YDataLogger 308

get\_autoStart, YWatchdog 1722

get\_averageValue, YDataRun 335

get\_averageValue, YDataStream 358

get\_averageValue, YMeasure 823

get\_baudRate, YHubPort 653

get\_beacon, YModule 841

get\_beaconDriven, YDataLogger 309

get\_bearing, YRefFrame 1266

get\_bitDirection, YDigitalIO 379

get\_bitOpenDrain, YDigitalIO 380

get\_bitPolarity, YDigitalIO 381

get\_bitState, YDigitalIO 382

get\_blinking, YLed 717

get\_brakingForce, YMotor 884

get\_brightness, YDisplay 424

get\_calibratedValue, YAnButton 123

get\_calibrationMax, YAnButton 124

get\_calibrationMin, YAnButton 125

get\_callbackCredentials, YNetwork 926

get\_callbackEncoding, YNetwork 927

get\_callbackMaxDelay, YNetwork 928

get\_callbackMethod, YNetwork 929

get\_callbackMinDelay, YNetwork 930

get\_callbackUrl, YNetwork 931

get\_channel, YWireless 1766

get\_columnCount, YDataStream 359

get\_columnNames, YDataStream 360

get\_cosPhi, YPower 1004

get\_countdown, YRelay 1297

get\_countdown, YWatchdog 1723

get\_CTS, YSerialPort 1371

get\_currentRawValue, YAccelerometer 39

get\_currentRawValue, YAltitude 81

get\_currentRawValue, YCarbonDioxide 161

get\_currentRawValue, YCompass 229

get\_currentRawValue, YCurrent 269

get\_currentRawValue, YGenericSensor 555

get\_currentRawValue, YGyro 604

get\_currentRawValue, YHumidity 680

get\_currentRawValue, YLightSensor 747

get\_currentRawValue, YMagnetometer 788

get\_currentRawValue, YPower 1005

get\_currentRawValue, YPressure 1047

get\_currentRawValue, YPwmInput 1086

get\_currentRawValue, YQt 1195

get\_currentRawValue, YSensor 1333

get\_currentRawValue, YTemperature 1464

get\_currentRawValue, YTilt 1505

get\_currentRawValue, YVoc 1544

get\_currentRawValue, YVoltage 1583

get\_currentRunIndex, YDataLogger 310

get\_currentValue, YAccelerometer 40

get\_currentValue, YAltitude 82

get\_currentValue, YCarbonDioxide 162

get\_currentValue, YCompass 230

get\_currentValue, YCurrent 270

get\_currentValue, YGenericSensor 556

get\_currentValue, YGyro 605

get\_currentValue, YHumidity 681

get\_currentValue, YLightSensor 748

get\_currentValue, YMagnetometer 789

get\_currentValue, YPower 1006  
get\_currentValue, YPressure 1048  
get\_currentValue, YPwmInput 1087  
get\_currentValue, YQt 1196  
get\_currentValue, YSensor 1334  
get\_currentValue, YTemperature 1465  
get\_currentValue, YTilt 1506  
get\_currentValue, YVoc 1545  
get\_currentValue, YVoltage 1584  
get\_cutOffVoltage, YMotor 885  
get\_data, YDataStream 361  
get\_dataRows, YDataStream 362  
get\_dataSamplesIntervalMs, YDataStream 363  
get\_dataSets, YDataLogger 311  
get\_dataStreams, YDataLogger 312  
get\_dateTime, YRealTimeClock 1232  
get\_detectedWlans, YWireless 1767  
get\_discoverable, YNetwork 932  
get\_display, YDisplayLayer 475  
get\_displayHeight, YDisplay 425  
get\_displayHeight, YDisplayLayer 476  
get\_displayLayer, YDisplay 426  
get\_displayType, YDisplay 427  
get\_displayWidth, YDisplay 428  
get\_displayWidth, YDisplayLayer 477  
get\_drivingForce, YMotor 886  
get\_duration, YDataRun 336  
get\_duration, YDataStream 364  
get\_dutyCycle, YPwmInput 1088  
get\_dutyCycle, YPwmOutput 1134  
get\_dutyCycleAtPowerOn, YPwmOutput 1135  
get\_enabled, YDisplay 429  
get\_enabled, YHubPort 654  
get\_enabled, YPwmOutput 1136  
get\_enabled, YServo 1428  
get\_enabledAtPowerOn, YPwmOutput 1137  
get\_enabledAtPowerOn, YServo 1429  
get\_endTimeUTC, YDataSet 346  
get\_endTimeUTC, YMeasure 824  
get\_errCount, YSerialPort 1373  
get\_errorMessage, YAccelerometer 41  
get\_errorMessage, YAltitude 83  
get\_errorMessage, YAnButton 126  
get\_errorMessage, YCarbonDioxide 163  
get\_errorMessage, YColorLed 198  
get\_errorMessage, YCompass 231  
get\_errorMessage, YCurrent 271  
get\_errorMessage, YDataLogger 313  
get\_errorMessage, YDigitalIO 383  
get\_errorMessage, YDisplay 430  
get\_errorMessage, YDualPower 500  
get\_errorMessage, YFiles 527  
get\_errorMessage, YGenericSensor 557  
get\_errorMessage, YGyro 606  
get\_errorMessage, YHubPort 655  
get\_errorMessage, YHumidity 682  
get\_errorMessage, YLed 718  
get\_errorMessage, YLightSensor 749  
get\_errorMessage, YMagnetometer 790  
get\_errorMessage, YModule 842  
get\_errorMessage, YMotor 887  
get\_errorMessage, YNetwork 933  
get\_errorMessage, YOsControl 979  
get\_errorMessage, YPower 1007  
get\_errorMessage, YPressure 1049  
get\_errorMessage, YPwmInput 1089  
get\_errorMessage, YPwmOutput 1138  
get\_errorMessage, YPwmPowerSource 1170  
get\_errorMessage, YQt 1197  
get\_errorMessage, YRealTimeClock 1233  
get\_errorMessage, YRefFrame 1267  
get\_errorMessage, YRelay 1298  
get\_errorMessage, YSensor 1335  
get\_errorMessage, YSerialPort 1374  
get\_errorMessage, YServo 1430  
get\_errorMessage, YTemperature 1466  
get\_errorMessage, YTilt 1507  
get\_errorMessage, YVoc 1546  
get\_errorMessage, YVoltage 1585  
get\_errorMessage, YVSource 1619  
get\_errorMessage, YWakeUpMonitor 1649  
get\_errorMessage, YWakeUpSchedule 1684  
get\_errorMessage, YWatchdog 1724  
get\_errorMessage, YWireless 1768  
get\_errorType, YAccelerometer 42  
get\_errorType, YAltitude 84  
get\_errorType, YAnButton 127  
get\_errorType, YCarbonDioxide 164  
get\_errorType, YColorLed 199  
get\_errorType, YCompass 232  
get\_errorType, YCurrent 272  
get\_errorType, YDataLogger 314  
get\_errorType, YDigitalIO 384  
get\_errorType, YDisplay 431  
get\_errorType, YDualPower 501  
get\_errorType, YFiles 528  
get\_errorType, YGenericSensor 558  
get\_errorType, YGyro 607  
get\_errorType, YHubPort 656  
get\_errorType, YHumidity 683  
get\_errorType, YLed 719  
get\_errorType, YLightSensor 750  
get\_errorType, YMagnetometer 791  
get\_errorType, YModule 843  
get\_errorType, YMotor 888  
get\_errorType, YNetwork 934  
get\_errorType, YOsControl 980  
get\_errorType, YPower 1008  
get\_errorType, YPressure 1050  
get\_errorType, YPwmInput 1090  
get\_errorType, YPwmOutput 1139  
get\_errorType, YPwmPowerSource 1171  
get\_errorType, YQt 1198  
get\_errorType, YRealTimeClock 1234  
get\_errorType, YRefFrame 1268  
get\_errorType, YRelay 1299  
get\_errorType, YSensor 1336  
get\_errorType, YSerialPort 1375

get\_errorType, YServo 1431  
get\_errorType, YTemperature 1467  
get\_errorType, YTilt 1508  
get\_errorType, YVoc 1547  
get\_errorType, YVoltage 1586  
get\_errorType, YVSource 1620  
get\_errorType, YWakeUpMonitor 1650  
get\_errorType, YWakeUpSchedule 1685  
get\_errorType, YWatchdog 1725  
get\_errorType, YWireless 1769  
get\_extPowerFailure, YVSource 1621  
get\_extVoltage, YDualPower 502  
get\_failSafeTimeout, YMotor 889  
get\_failure, YVSource 1622  
get\_filesCount, YFiles 529  
get\_firmwareRelease, YModule 844  
get\_freeSpace, YFiles 530  
get\_frequency, YMotor 890  
get\_frequency, YPwmInput 1091  
get\_frequency, YPwmOutput 1140  
get\_friendlyName, YAccelerometer 43  
get\_friendlyName, YAltitude 85  
get\_friendlyName, YAnButton 128  
get\_friendlyName, YCarbonDioxide 165  
get\_friendlyName, YColorLed 200  
get\_friendlyName, YCompass 233  
get\_friendlyName, YCurrent 273  
get\_friendlyName, YDataLogger 315  
get\_friendlyName, YDigitalIO 385  
get\_friendlyName, YDisplay 432  
get\_friendlyName, YDualPower 503  
get\_friendlyName, YFiles 531  
get\_friendlyName, YGenericSensor 559  
get\_friendlyName, YGyro 608  
get\_friendlyName, YHubPort 657  
get\_friendlyName, YHumidity 684  
get\_friendlyName, YLed 720  
get\_friendlyName, YLightSensor 751  
get\_friendlyName, YMagnetometer 792  
get\_friendlyName, YMotor 891  
get\_friendlyName, YNetwork 935  
get\_friendlyName, YOsControl 981  
get\_friendlyName, YPower 1009  
get\_friendlyName, YPressure 1051  
get\_friendlyName, YPwmInput 1092  
get\_friendlyName, YPwmOutput 1141  
get\_friendlyName, YPwmPowerSource 1172  
get\_friendlyName, YQt 1199  
get\_friendlyName, YRealTimeClock 1235  
get\_friendlyName, YRefFrame 1269  
get\_friendlyName, YRelay 1300  
get\_friendlyName, YSensor 1337  
get\_friendlyName, YSerialPort 1376  
get\_friendlyName, YServo 1432  
get\_friendlyName, YTemperature 1468  
get\_friendlyName, YTilt 1509  
get\_friendlyName, YVoc 1548  
get\_friendlyName, YVoltage 1587  
get\_friendlyName, YWakeUpMonitor 1651  
get\_friendlyName, YWakeUpSchedule 1686  
get\_friendlyName, YWatchdog 1726  
get\_friendlyName, YWireless 1770  
get\_functionDescriptor, YAccelerometer 44  
get\_functionDescriptor, YAltitude 86  
get\_functionDescriptor, YAnButton 129  
get\_functionDescriptor, YCarbonDioxide 166  
get\_functionDescriptor, YColorLed 201  
get\_functionDescriptor, YCompass 234  
get\_functionDescriptor, YCurrent 274  
get\_functionDescriptor, YDataLogger 316  
get\_functionDescriptor, YDigitalIO 386  
get\_functionDescriptor, YDisplay 433  
get\_functionDescriptor, YDualPower 504  
get\_functionDescriptor, YFiles 532  
get\_functionDescriptor, YGenericSensor 560  
get\_functionDescriptor, YGyro 609  
get\_functionDescriptor, YHubPort 658  
get\_functionDescriptor, YHumidity 685  
get\_functionDescriptor, YLed 721  
get\_functionDescriptor, YLightSensor 752  
get\_functionDescriptor, YMagnetometer 793  
get\_functionDescriptor, YMotor 892  
get\_functionDescriptor, YNetwork 936  
get\_functionDescriptor, YOsControl 982  
get\_functionDescriptor, YPower 1010  
get\_functionDescriptor, YPressure 1052  
get\_functionDescriptor, YPwmInput 1093  
get\_functionDescriptor, YPwmOutput 1142  
get\_functionDescriptor, YPwmPowerSource 1173  
get\_functionDescriptor, YQt 1200  
get\_functionDescriptor, YRealTimeClock 1236  
get\_functionDescriptor, YRefFrame 1270  
get\_functionDescriptor, YRelay 1301  
get\_functionDescriptor, YSensor 1338  
get\_functionDescriptor, YSerialPort 1377  
get\_functionDescriptor, YServo 1433  
get\_functionDescriptor, YTemperature 1469  
get\_functionDescriptor, YTilt 1510  
get\_functionDescriptor, YVoc 1549  
get\_functionDescriptor, YVoltage 1588  
get\_functionDescriptor, YVSource 1623  
get\_functionDescriptor, YWakeUpMonitor 1652  
get\_functionDescriptor, YWakeUpSchedule 1687  
get\_functionDescriptor, YWatchdog 1727  
get\_functionDescriptor, YWireless 1771  
get\_functionId, YAccelerometer 45  
get\_functionId, YAltitude 87  
get\_functionId, YAnButton 130  
get\_functionId, YCarbonDioxide 167  
get\_functionId, YColorLed 202  
get\_functionId, YCompass 235  
get\_functionId, YCurrent 275  
get\_functionId, YDataLogger 317  
get\_functionId, YDataSet 347  
get\_functionId, YDigitalIO 387  
get\_functionId, YDisplay 434  
get\_functionId, YDualPower 505  
get\_functionId, YFiles 533

get\_functionId, YGenericSensor 561  
get\_functionId, YGyro 610  
get\_functionId, YHubPort 659  
get\_functionId, YHumidity 686  
get\_functionId, YLed 722  
get\_functionId, YLightSensor 753  
get\_functionId, YMagnetometer 794  
get\_functionId, YMotor 893  
get\_functionId, YNetwork 937  
get\_functionId, YOsControl 983  
get\_functionId, YPower 1011  
get\_functionId, YPressure 1053  
get\_functionId, YPwmInput 1094  
get\_functionId, YPwmOutput 1143  
get\_functionId, YPwmPowerSource 1174  
get\_functionId, YQt 1201  
get\_functionId, YRealTimeClock 1237  
get\_functionId, YRefFrame 1271  
get\_functionId, YRelay 1302  
get\_functionId, YSensor 1339  
get\_functionId, YSerialPort 1378  
get\_functionId, YServo 1434  
get\_functionId, YTemperature 1470  
get\_functionId, YTilt 1511  
get\_functionId, YVoc 1550  
get\_functionId, YVoltage 1589  
get\_functionId, YWakeUpMonitor 1653  
get\_functionId, YWakeUpSchedule 1688  
get\_functionId, YWatchdog 1728  
get\_functionId, YWireless 1772  
get\_hardwareId, YAccelerometer 46  
get\_hardwareId, YAltitude 88  
get\_hardwareId, YAnButton 131  
get\_hardwareId, YCarbonDioxide 168  
get\_hardwareId, YColorLed 203  
get\_hardwareId, YCompass 236  
get\_hardwareId, YCurrent 276  
get\_hardwareId, YDataLogger 318  
get\_hardwareId, YDataSet 348  
get\_hardwareId, YDigitalIO 388  
get\_hardwareId, YDisplay 435  
get\_hardwareId, YDualPower 506  
get\_hardwareId, YFiles 534  
get\_hardwareId, YGenericSensor 562  
get\_hardwareId, YGyro 611  
get\_hardwareId, YHubPort 660  
get\_hardwareId, YHumidity 687  
get\_hardwareId, YLed 723  
get\_hardwareId, YLightSensor 754  
get\_hardwareId, YMagnetometer 795  
get\_hardwareId, YModule 845  
get\_hardwareId, YMotor 894  
get\_hardwareId, YNetwork 938  
get\_hardwareId, YOsControl 984  
get\_hardwareId, YPower 1012  
get\_hardwareId, YPressure 1054  
get\_hardwareId, YPwmInput 1095  
get\_hardwareId, YPwmOutput 1144  
get\_hardwareId, YPwmPowerSource 1175  
get\_hardwareId, YQt 1202  
get\_hardwareId, YRealTimeClock 1238  
get\_hardwareId, YRefFrame 1272  
get\_hardwareId, YRelay 1303  
get\_hardwareId, YSensor 1340  
get\_hardwareId, YSerialPort 1379  
get\_hardwareId, YServo 1435  
get\_hardwareId, YTemperature 1471  
get\_hardwareId, YTilt 1512  
get\_hardwareId, YVoc 1551  
get\_hardwareId, YVoltage 1590  
get\_hardwareId, YWakeUpMonitor 1654  
get\_hardwareId, YWakeUpSchedule 1689  
get\_hardwareId, YWatchdog 1729  
get\_hardwareId, YWireless 1773  
get\_heading, YGyro 612  
get\_highestValue, YAccelerometer 47  
get\_highestValue, YAltitude 89  
get\_highestValue, YCarbonDioxide 169  
get\_highestValue, YCompass 237  
get\_highestValue, YCurrent 277  
get\_highestValue, YGenericSensor 563  
get\_highestValue, YGyro 613  
get\_highestValue, YHumidity 688  
get\_highestValue, YLightSensor 755  
get\_highestValue, YMagnetometer 796  
get\_highestValue, YPower 1013  
get\_highestValue, YPressure 1055  
get\_highestValue, YPwmInput 1096  
get\_highestValue, YQt 1203  
get\_highestValue, YSensor 1341  
get\_highestValue, YTemperature 1472  
get\_highestValue, YTilt 1513  
get\_highestValue, YVoc 1552  
get\_highestValue, YVoltage 1591  
get\_hours, YWakeUpSchedule 1690  
get\_hslColor, YColorLed 204  
get\_icon2d, YModule 846  
get\_ipAddress, YNetwork 939  
get\_isPressed, YAnButton 132  
get\_lastLogs, YModule 847  
get\_lastMsg, YSerialPort 1380  
get\_lastTimePressed, YAnButton 133  
get\_lastTimeReleased, YAnButton 134  
get\_layerCount, YDisplay 436  
get\_layerHeight, YDisplay 437  
get\_layerHeight, YDisplayLayer 478  
get\_layerWidth, YDisplay 438  
get\_layerWidth, YDisplayLayer 479  
get\_linkQuality, YWireless 1774  
get\_list, YFiles 535  
get\_logFrequency, YAccelerometer 48  
get\_logFrequency, YAltitude 90  
get\_logFrequency, YCarbonDioxide 170  
get\_logFrequency, YCompass 238  
get\_logFrequency, YCurrent 278  
get\_logFrequency, YGenericSensor 564  
get\_logFrequency, YGyro 614  
get\_logFrequency, YHumidity 689



get\_logFrequency, YLightSensor 756  
get\_logFrequency, YMagnetometer 797  
get\_logFrequency, YPower 1014  
get\_logFrequency, YPressure 1056  
get\_logFrequency, YPwmInput 1097  
get\_logFrequency, YQt 1204  
get\_logFrequency, YSensor 1342  
get\_logFrequency, YTemperature 1473  
get\_logFrequency, YTilt 1514  
get\_logFrequency, YVoc 1553  
get\_logFrequency, YVoltage 1592  
get\_logicalName, YAccelerometer 49  
get\_logicalName, YAltitude 91  
get\_logicalName, YAnButton 135  
get\_logicalName, YCarbonDioxide 171  
get\_logicalName, YColorLed 205  
get\_logicalName, YCompass 239  
get\_logicalName, YCurrent 279  
get\_logicalName, YDataLogger 319  
get\_logicalName, YDigitalIO 389  
get\_logicalName, YDisplay 439  
get\_logicalName, YDualPower 507  
get\_logicalName, YFiles 536  
get\_logicalName, YGenericSensor 565  
get\_logicalName, YGyro 615  
get\_logicalName, YHubPort 661  
get\_logicalName, YHumidity 690  
get\_logicalName, YLed 724  
get\_logicalName, YLightSensor 757  
get\_logicalName, YMagnetometer 798  
get\_logicalName, YModule 848  
get\_logicalName, YMotor 895  
get\_logicalName, YNetwork 940  
get\_logicalName, YOsControl 985  
get\_logicalName, YPower 1015  
get\_logicalName, YPressure 1057  
get\_logicalName, YPwmInput 1098  
get\_logicalName, YPwmOutput 1145  
get\_logicalName, YPwmPowerSource 1176  
get\_logicalName, YQt 1205  
get\_logicalName, YRealTimeClock 1239  
get\_logicalName, YRefFrame 1273  
get\_logicalName, YRelay 1304  
get\_logicalName, YSensor 1343  
get\_logicalName, YSerialPort 1381  
get\_logicalName, YServo 1436  
get\_logicalName, YTemperature 1474  
get\_logicalName, YTilt 1515  
get\_logicalName, YVoc 1554  
get\_logicalName, YVoltage 1593  
get\_logicalName, YVSource 1624  
get\_logicalName, YWakeUpMonitor 1655  
get\_logicalName, YWakeUpSchedule 1691  
get\_logicalName, YWatchdog 1730  
get\_logicalName, YWireless 1775  
get\_lowestValue, YAccelerometer 50  
get\_lowestValue, YAltitude 92  
get\_lowestValue, YCarbonDioxide 172  
get\_lowestValue, YCompass 240

get\_lowestValue, YCurrent 280  
get\_lowestValue, YGenericSensor 566  
get\_lowestValue, YGyro 616  
get\_lowestValue, YHumidity 691  
get\_lowestValue, YLightSensor 758  
get\_lowestValue, YMagnetometer 799  
get\_lowestValue, YPower 1016  
get\_lowestValue, YPressure 1058  
get\_lowestValue, YPwmInput 1099  
get\_lowestValue, YQt 1206  
get\_lowestValue, YSensor 1344  
get\_lowestValue, YTemperature 1475  
get\_lowestValue, YTilt 1516  
get\_lowestValue, YVoc 1555  
get\_lowestValue, YVoltage 1594  
get\_luminosity, YLed 725  
get\_luminosity, YModule 849  
get\_macAddress, YNetwork 941  
get\_magneticHeading, YCompass 241  
get\_maxTimeOnStateA, YRelay 1305  
get\_maxTimeOnStateA, YWatchdog 1731  
get\_maxTimeOnStateB, YRelay 1306  
get\_maxTimeOnStateB, YWatchdog 1732  
get\_maxValue, YDataRun 337  
get\_maxValue, YDataStream 365  
get\_maxValue, YMeasure 825  
get\_measureNames, YDataRun 338  
get\_measures, YDataSet 349  
get\_measureType, YLightSensor 759  
get\_message, YWireless 1776  
get\_meter, YPower 1017  
get\_meterTimer, YPower 1018  
get\_minutes, YWakeUpSchedule 1692  
get\_minutesA, YWakeUpSchedule 1693  
get\_minutesB, YWakeUpSchedule 1694  
get\_minValue, YDataRun 339  
get\_minValue, YDataStream 366  
get\_minValue, YMeasure 826  
get\_module, YAccelerometer 51  
get\_module, YAltitude 93  
get\_module, YAnButton 136  
get\_module, YCarbonDioxide 173  
get\_module, YColorLed 206  
get\_module, YCompass 242  
get\_module, YCurrent 281  
get\_module, YDataLogger 320  
get\_module, YDigitalIO 390  
get\_module, YDisplay 440  
get\_module, YDualPower 508  
get\_module, YFiles 537  
get\_module, YGenericSensor 567  
get\_module, YGyro 617  
get\_module, YHubPort 662  
get\_module, YHumidity 692  
get\_module, YLed 726  
get\_module, YLightSensor 760  
get\_module, YMagnetometer 800  
get\_module, YMotor 896  
get\_module, YNetwork 942

get\_module, YOsControl 986  
get\_module, YPower 1019  
get\_module, YPressure 1059  
get\_module, YPwmInput 1100  
get\_module, YPwmOutput 1146  
get\_module, YPwmPowerSource 1177  
get\_module, YQt 1207  
get\_module, YRealTimeClock 1240  
get\_module, YRefFrame 1274  
get\_module, YRelay 1307  
get\_module, YSensor 1345  
get\_module, YSerialPort 1382  
get\_module, YServo 1437  
get\_module, YTemperature 1476  
get\_module, YTilt 1517  
get\_module, YVoc 1556  
get\_module, YVoltage 1595  
get\_module, YVSource 1625  
get\_module, YWakeUpMonitor 1656  
get\_module, YWakeUpSchedule 1695  
get\_module, YWatchdog 1733  
get\_module, YWireless 1777  
get\_monthDays, YWakeUpSchedule 1696  
get\_months, YWakeUpSchedule 1697  
get\_motorStatus, YMotor 897  
get\_mountOrientation, YRefFrame 1275  
get\_mountPosition, YRefFrame 1276  
get\_msgCount, YSerialPort 1383  
get\_neutral, YServo 1438  
get\_nextOccurence, YWakeUpSchedule 1698  
get\_nextWakeUp, YWakeUpMonitor 1657  
get\_orientation, YDisplay 441  
get\_output, YRelay 1308  
get\_output, YWatchdog 1734  
get\_outputVoltage, YDigitalIO 391  
get\_overCurrent, YVSource 1626  
get\_overCurrentLimit, YMotor 898  
get\_overHeat, YVSource 1627  
get\_overLoad, YVSource 1628  
get\_period, YPwmInput 1101  
get\_period, YPwmOutput 1147  
get\_persistentSettings, YModule 850  
get\_pitch, YGyro 618  
get\_poeCurrent, YNetwork 943  
get\_portDirection, YDigitalIO 392  
get\_portOpenDrain, YDigitalIO 393  
get\_portPolarity, YDigitalIO 394  
get\_portSize, YDigitalIO 395  
get\_portState, YDigitalIO 396  
get\_portState, YHubPort 663  
get\_position, YServo 1439  
get\_positionAtPowerOn, YServo 1440  
get\_power, YLed 727  
get\_powerControl, YDualPower 509  
get\_powerDuration, YWakeUpMonitor 1658  
get\_powerMode, YPwmPowerSource 1178  
get\_powerState, YDualPower 510  
get\_preview, YDataSet 350  
get\_primaryDNS, YNetwork 944  
get\_productId, YModule 851  
get\_productName, YModule 852  
get\_productRelease, YModule 853  
get\_progress, YDataSet 351  
get\_protocol, YSerialPort 1384  
get\_pulseCounter, YAnButton 137  
get\_pulseCounter, YPwmInput 1102  
get\_pulseDuration, YPwmInput 1103  
get\_pulseDuration, YPwmOutput 1148  
get\_pulseTimer, YAnButton 138  
get\_pulseTimer, YPwmInput 1104  
get\_pulseTimer, YRelay 1309  
get\_pulseTimer, YWatchdog 1735  
get\_pwmReportMode, YPwmInput 1105  
get\_qnh, YAltitude 94  
get\_quaternionW, YGyro 619  
get\_quaternionX, YGyro 620  
get\_quaternionY, YGyro 621  
get\_quaternionZ, YGyro 622  
get\_range, YServo 1441  
get\_rawValue, YAnButton 139  
get\_readiness, YNetwork 945  
get\_rebootCountdown, YModule 854  
get\_recordedData, YAccelerometer 52  
get\_recordedData, YAltitude 95  
get\_recordedData, YCarbonDioxide 174  
get\_recordedData, YCompass 243  
get\_recordedData, YCurrent 282  
get\_recordedData, YGenericSensor 568  
get\_recordedData, YGyro 623  
get\_recordedData, YHumidity 693  
get\_recordedData, YLightSensor 761  
get\_recordedData, YMagnetometer 801  
get\_recordedData, YPower 1020  
get\_recordedData, YPressure 1060  
get\_recordedData, YPwmInput 1106  
get\_recordedData, YQt 1208  
get\_recordedData, YSensor 1346  
get\_recordedData, YTemperature 1477  
get\_recordedData, YTilt 1518  
get\_recordedData, YVoc 1557  
get\_recordedData, YVoltage 1596  
get\_recording, YDataLogger 321  
get\_regulationFailure, YVSource 1629  
get\_reportFrequency, YAccelerometer 53  
get\_reportFrequency, YAltitude 96  
get\_reportFrequency, YCarbonDioxide 175  
get\_reportFrequency, YCompass 244  
get\_reportFrequency, YCurrent 283  
get\_reportFrequency, YGenericSensor 569  
get\_reportFrequency, YGyro 624  
get\_reportFrequency, YHumidity 694  
get\_reportFrequency, YLightSensor 762  
get\_reportFrequency, YMagnetometer 802  
get\_reportFrequency, YPower 1021  
get\_reportFrequency, YPressure 1061  
get\_reportFrequency, YPwmInput 1107  
get\_reportFrequency, YQt 1209  
get\_reportFrequency, YSensor 1347

get\_reportFrequency, YTemperature 1478  
get\_reportFrequency, YTilt 1519  
get\_reportFrequency, YVoc 1558  
get\_reportFrequency, YVoltage 1597  
get\_resolution, YAccelerometer 54  
get\_resolution, YAltitude 97  
get\_resolution, YCarbonDioxide 176  
get\_resolution, YCompass 245  
get\_resolution, YCurrent 284  
get\_resolution, YGenericSensor 570  
get\_resolution, YGyro 625  
get\_resolution, YHumidity 695  
get\_resolution, YLightSensor 763  
get\_resolution, YMagnetometer 803  
get\_resolution, YPower 1022  
get\_resolution, YPressure 1062  
get\_resolution, YPwmInput 1108  
get\_resolution, YQt 1210  
get\_resolution, YSensor 1348  
get\_resolution, YTemperature 1479  
get\_resolution, YTilt 1520  
get\_resolution, YVoc 1559  
get\_resolution, YVoltage 1598  
get\_rgbColor, YColorLed 207  
get\_rgbColorAtPowerOn, YColorLed 208  
get\_roll, YGyro 626  
get\_router, YNetwork 946  
get\_rowCount, YDataStream 367  
get\_runIndex, YDataStream 368  
get\_running, YWatchdog 1736  
get\_rxCount, YSerialPort 1385  
get\_secondaryDNS, YNetwork 947  
get\_security, YWireless 1778  
get\_sensitivity, YAnButton 140  
get\_sensorType, YTemperature 1480  
get\_serialMode, YSerialPort 1386  
get\_serialNumber, YModule 855  
get\_shutdownCountdown, YOsControl 987  
get\_signalBias, YGenericSensor 571  
get\_signalRange, YGenericSensor 572  
get\_signalUnit, YGenericSensor 573  
get\_signalValue, YGenericSensor 574  
get\_sleepCountdown, YWakeUpMonitor 1659  
get\_ssid, YWireless 1779  
get\_starterTime, YMotor 899  
get\_startTime, YDataStream 369  
get\_startTimeUTC, YDataRun 340  
get\_startTimeUTC, YDataSet 352  
get\_startTimeUTC, YDataStream 370  
get\_startTimeUTC, YMeasure 827  
get\_startupSeq, YDisplay 442  
get\_state, YRelay 1310  
get\_state, YWatchdog 1737  
get\_stateAtPowerOn, YRelay 1311  
get\_stateAtPowerOn, YWatchdog 1738  
get\_subnetMask, YNetwork 948  
get\_summary, YDataSet 353  
get\_timeSet, YRealTimeClock 1241  
get\_timeUTC, YDataLogger 322  
get\_triggerDelay, YWatchdog 1739  
get\_triggerDuration, YWatchdog 1740  
get\_txCount, YSerialPort 1387  
get\_unit, YAccelerometer 55  
get\_unit, YAltitude 98  
get\_unit, YCarbonDioxide 177  
get\_unit, YCompass 246  
get\_unit, YCurrent 285  
get\_unit, YDataSet 354  
get\_unit, YGenericSensor 575  
get\_unit, YGyro 627  
get\_unit, YHumidity 696  
get\_unit, YLightSensor 764  
get\_unit, YMagnetometer 804  
get\_unit, YPower 1023  
get\_unit, YPressure 1063  
get\_unit, YPwmInput 1109  
get\_unit, YQt 1211  
get\_unit, YSensor 1349  
get\_unit, YTemperature 1481  
get\_unit, YTilt 1521  
get\_unit, YVoc 1560  
get\_unit, YVoltage 1599  
get\_unit, YVSource 1630  
get\_unixTime, YRealTimeClock 1242  
get\_upTime, YModule 856  
get\_usbCurrent, YModule 857  
get\_userData, YAccelerometer 56  
get\_userData, YAltitude 99  
get\_userData, YAnButton 141  
get\_userData, YCarbonDioxide 178  
get\_userData, YColorLed 209  
get\_userData, YCompass 247  
get\_userData, YCurrent 286  
get\_userData, YDataLogger 323  
get\_userData, YDigitalIO 397  
get\_userData, YDisplay 443  
get\_userData, YDualPower 511  
get\_userData, YFiles 538  
get\_userData, YGenericSensor 576  
get\_userData, YGyro 628  
get\_userData, YHubPort 664  
get\_userData, YHumidity 697  
get\_userData, YLed 728  
get\_userData, YLightSensor 765  
get\_userData, YMagnetometer 805  
get\_userData, YModule 858  
get\_userData, YMotor 900  
get\_userData, YNetwork 949  
get\_userData, YOsControl 988  
get\_userData, YPower 1024  
get\_userData, YPressure 1064  
get\_userData, YPwmInput 1110  
get\_userData, YPwmOutput 1149  
get\_userData, YPwmPowerSource 1179  
get\_userData, YQt 1212  
get\_userData, YRealTimeClock 1243  
get\_userData, YRefFrame 1277  
get\_userData, YRelay 1312

get\_userdata, YSensor 1350  
get\_userdata, YSerialPort 1388  
get\_userdata, YServo 1442  
get\_userdata, YTemperature 1482  
get\_userdata, YTilt 1522  
get\_userdata, YVoc 1561  
get\_userdata, YVoltage 1600  
get\_userdata, YVSource 1631  
get\_userdata, YWakeUpMonitor 1660  
get\_userdata, YWakeUpSchedule 1699  
get\_userdata, YWatchdog 1741  
get\_userdata, YWireless 1780  
get\_userPassword, YNetwork 950  
get\_userVar, YModule 859  
get\_utcOffset, YRealTimeClock 1244  
get\_valueCount, YDataRun 341  
get\_valueInterval, YDataRun 342  
get\_valueRange, YGenericSensor 577  
get\_voltage, YVSource 1632  
get\_wakeUpReason, YWakeUpMonitor 1661  
get\_wakeUpState, YWakeUpMonitor 1662  
get\_weekDays, YWakeUpSchedule 1700  
get\_wwwWatchdogDelay, YNetwork 951  
get\_xValue, YAccelerometer 57  
get\_xValue, YGyro 629  
get\_xValue, YMagnetometer 806  
get\_yValue, YAccelerometer 58  
get\_yValue, YGyro 630  
get\_yValue, YMagnetometer 807  
get\_zValue, YAccelerometer 59  
get\_zValue, YGyro 631  
get\_zValue, YMagnetometer 808  
GetAPIVersion, YAPI 16  
GetTickCount, YAPI 17  
Gyroscope 597

## H

HandleEvents, YAPI 18  
hide, YDisplayLayer 480  
hslMove, YColorLed 210  
Humidity 673

## I

InitAPI, YAPI 19  
Interface 32, 74, 116, 154, 193, 222, 262, 301,  
372, 416, 463, 495, 520, 548, 597, 648, 673,  
712, 739, 781, 829, 876, 917, 997, 1040, 1079,  
1127, 1165, 1188, 1227, 1290, 1326, 1365,  
1422, 1457, 1498, 1537, 1576, 1615, 1643,  
1678, 1715, 1760  
Introduction 1  
isOnline, YAccelerometer 60  
isOnline, YAltitude 100  
isOnline, YAnButton 142  
isOnline, YCarbonDioxide 179  
isOnline, YColorLed 211  
isOnline, YCompass 248  
isOnline, YCurrent 287

isOnline, YDataLogger 324  
isOnline, YDigitalIO 398  
isOnline, YDisplay 444  
isOnline, YDualPower 512  
isOnline, YFiles 539  
isOnline, YGenericSensor 578  
isOnline, YGyro 632  
isOnline, YHubPort 665  
isOnline, YHumidity 698  
isOnline, YLed 729  
isOnline, YLightSensor 766  
isOnline, YMagnetometer 809  
isOnline, YModule 860  
isOnline, YMotor 901  
isOnline, YNetwork 952  
isOnline, YOsControl 989  
isOnline, YPower 1025  
isOnline, YPressure 1065  
isOnline, YPwmInput 1111  
isOnline, YPwmOutput 1150  
isOnline, YPwmPowerSource 1180  
isOnline, YQt 1213  
isOnline, YRealTimeClock 1245  
isOnline, YRefFrame 1278  
isOnline, YRelay 1313  
isOnline, YSensor 1351  
isOnline, YSerialPort 1389  
isOnline, YServo 1443  
isOnline, YTemperature 1483  
isOnline, YTilt 1523  
isOnline, YVoc 1562  
isOnline, YVoltage 1601  
isOnline, YVSource 1633  
isOnline, YWakeUpMonitor 1663  
isOnline, YWakeUpSchedule 1701  
isOnline, YWatchdog 1742  
isOnline, YWireless 1781

## J

joinNetwork, YWireless 1782

## K

keepALive, YMotor 902

## L

Library 3  
LightSensor 739  
lineTo, YDisplayLayer 481  
load, YAccelerometer 61  
load, YAltitude 101  
load, YAnButton 143  
load, YCarbonDioxide 180  
load, YColorLed 212  
load, YCompass 249  
load, YCurrent 288  
load, YDataLogger 325  
load, YDigitalIO 399

load, YDisplay 445  
 load, YDualPower 513  
 load, YFiles 540  
 load, YGenericSensor 579  
 load, YGyro 633  
 load, YHubPort 666  
 load, YHumidity 699  
 load, YLed 730  
 load, YLightSensor 767  
 load, YMagnetometer 810  
 load, YModule 861  
 load, YMotor 903  
 load, YNetwork 953  
 load, YOsControl 990  
 load, YPower 1026  
 load, YPressure 1066  
 load, YPwmInput 1112  
 load, YPwmOutput 1151  
 load, YPwmPowerSource 1181  
 load, YQt 1214  
 load, YRealTimeClock 1246  
 load, YRefFrame 1279  
 load, YRelay 1314  
 load, YSensor 1352  
 load, YSerialPort 1390  
 load, YServo 1444  
 load, YTemperature 1484  
 load, YTilt 1524  
 load, YVoc 1563  
 load, YVoltage 1602  
 load, YVSource 1634  
 load, YWakeUpMonitor 1664  
 load, YWakeUpSchedule 1702  
 load, YWatchdog 1743  
 load, YWireless 1783  
 loadCalibrationPoints, YAccelerometer 62  
 loadCalibrationPoints, YAltitude 102  
 loadCalibrationPoints, YCarbonDioxide 181  
 loadCalibrationPoints, YCompass 250  
 loadCalibrationPoints, YCurrent 289  
 loadCalibrationPoints, YGenericSensor 580  
 loadCalibrationPoints, YGyro 634  
 loadCalibrationPoints, YHumidity 700  
 loadCalibrationPoints, YLightSensor 768  
 loadCalibrationPoints, YMagnetometer 811  
 loadCalibrationPoints, YPower 1027  
 loadCalibrationPoints, YPressure 1067  
 loadCalibrationPoints, YPwmInput 1113  
 loadCalibrationPoints, YQt 1215  
 loadCalibrationPoints, YSensor 1353  
 loadCalibrationPoints, YTemperature 1485  
 loadCalibrationPoints, YTilt 1525  
 loadCalibrationPoints, YVoc 1564  
 loadCalibrationPoints, YVoltage 1603  
 loadMore, YDataSet 355

## M

Magnetometer 781  
 Measured 823

modbusReadBits, YSerialPort 1391  
 modbusReadInputBits, YSerialPort 1392  
 modbusReadInputRegisters, YSerialPort 1393  
 modbusReadRegisters, YSerialPort 1394  
 modbusWriteAndReadRegisters, YSerialPort  
 1395  
 modbusWriteBit, YSerialPort 1396  
 modbusWriteBits, YSerialPort 1397  
 modbusWriteRegister, YSerialPort 1398  
 modbusWriteRegisters, YSerialPort 1399  
 Module 5, 829  
 more3DCalibration, YRefFrame 1280  
 Motor 876  
 move, YServo 1445  
 moveTo, YDisplayLayer 482

## N

Network 917  
 newSequence, YDisplay 446  
 nextAccelerometer, YAccelerometer 63  
 nextAltitude, YAltitude 103  
 nextAnButton, YAnButton 144  
 nextCarbonDioxide, YCarbonDioxide 182  
 nextColorLed, YColorLed 213  
 nextCompass, YCompass 251  
 nextCurrent, YCurrent 290  
 nextDataLogger, YDataLogger 326  
 nextDigitalIO, YDigitalIO 400  
 nextDisplay, YDisplay 447  
 nextDualPower, YDualPower 514  
 nextFiles, YFiles 541  
 nextGenericSensor, YGenericSensor 581  
 nextGyro, YGyro 635  
 nextHubPort, YHubPort 667  
 nextHumidity, YHumidity 701  
 nextLed, YLed 731  
 nextLightSensor, YLightSensor 769  
 nextMagnetometer, YMagnetometer 812  
 nextModule, YModule 862  
 nextMotor, YMotor 904  
 nextNetwork, YNetwork 954  
 nextOsControl, YOsControl 991  
 nextPower, YPower 1028  
 nextPressure, YPressure 1068  
 nextPwmInput, YPwmInput 1114  
 nextPwmOutput, YPwmOutput 1152  
 nextPwmPowerSource, YPwmPowerSource  
 1182  
 nextQt, YQt 1216  
 nextRealTimeClock, YRealTimeClock 1247  
 nextRefFrame, YRefFrame 1281  
 nextRelay, YRelay 1315  
 nextSensor, YSensor 1354  
 nextSerialPort, YSerialPort 1400  
 nextServo, YServo 1446  
 nextTemperature, YTemperature 1486  
 nextTilt, YTilt 1526  
 nextVoc, YVoc 1565  
 nextVoltage, YVoltage 1604

nextVSource, YVSource 1635  
nextWakeUpMonitor, YWakeUpMonitor 1665  
nextWakeUpSchedule, YWakeUpSchedule 1703  
nextWatchdog, YWatchdog 1744  
nextWireless, YWireless 1784

## O

Object 463

## P

pauseSequence, YDisplay 448  
ping, YNetwork 955  
playSequence, YDisplay 449  
Port 648  
Power 495, 997  
PreregisterHub, YAPI 20  
Pressure 1040  
pulse, YDigitalIO 401  
pulse, YRelay 1316  
pulse, YVSource 1636  
pulse, YWatchdog 1745  
pulseDurationMove, YPwmOutput 1153  
PwmInput 1079  
PwmPowerSource 1165  
Python 3

## Q

Quaternion 1188  
queryLine, YSerialPort 1401  
queryMODBUS, YSerialPort 1402

## R

read\_seek, YSerialPort 1407  
readHex, YSerialPort 1403  
readLine, YSerialPort 1404  
readMessages, YSerialPort 1405  
readStr, YSerialPort 1406  
Real 1227  
reboot, YModule 863  
Recorded 345  
Reference 10, 1254  
registerAnglesCallback, YGyro 636  
RegisterDeviceArrivalCallback, YAPI 21  
RegisterDeviceRemovalCallback, YAPI 22  
RegisterHub, YAPI 23  
RegisterHubDiscoveryCallback, YAPI 24  
registerLogCallback, YModule 864  
RegisterLogFunction, YAPI 25  
registerQuaternionCallback, YGyro 637  
registerTimedReportCallback, YAccelerometer 64  
registerTimedReportCallback, YAltitude 104  
registerTimedReportCallback, YCarbonDioxide 183  
registerTimedReportCallback, YCompass 252  
registerTimedReportCallback, YCurrent 291  
registerTimedReportCallback, YGenericSensor

582

registerTimedReportCallback, YGyro 638  
registerTimedReportCallback, YHumidity 702  
registerTimedReportCallback, YLightSensor 770  
registerTimedReportCallback, YMagnetometer 813  
registerTimedReportCallback, YPower 1029  
registerTimedReportCallback, YPressure 1069  
registerTimedReportCallback, YPwmInput 1115  
registerTimedReportCallback, YQt 1217  
registerTimedReportCallback, YSensor 1355  
registerTimedReportCallback, YTemperature 1487  
registerTimedReportCallback, YTilt 1527  
registerTimedReportCallback, YVoc 1566  
registerTimedReportCallback, YVoltage 1605  
registerValueCallback, YAccelerometer 65  
registerValueCallback, YAltitude 105  
registerValueCallback, YAnButton 145  
registerValueCallback, YCarbonDioxide 184  
registerValueCallback, YColorLed 214  
registerValueCallback, YCompass 253  
registerValueCallback, YCurrent 292  
registerValueCallback, YDataLogger 327  
registerValueCallback, YDigitalIO 402  
registerValueCallback, YDisplay 450  
registerValueCallback, YDualPower 515  
registerValueCallback, YFiles 542  
registerValueCallback, YGenericSensor 583  
registerValueCallback, YGyro 639  
registerValueCallback, YHubPort 668  
registerValueCallback, YHumidity 703  
registerValueCallback, YLed 732  
registerValueCallback, YLightSensor 771  
registerValueCallback, YMagnetometer 814  
registerValueCallback, YMotor 905  
registerValueCallback, YNetwork 956  
registerValueCallback, YOsControl 992  
registerValueCallback, YPower 1030  
registerValueCallback, YPressure 1070  
registerValueCallback, YPwmInput 1116  
registerValueCallback, YPwmOutput 1154  
registerValueCallback, YPwmPowerSource 1183  
registerValueCallback, YQt 1218  
registerValueCallback, YRealTimeClock 1248  
registerValueCallback, YRefFrame 1282  
registerValueCallback, YRelay 1317  
registerValueCallback, YSensor 1356  
registerValueCallback, YSerialPort 1408  
registerValueCallback, YServo 1447  
registerValueCallback, YTemperature 1488  
registerValueCallback, YTilt 1528  
registerValueCallback, YVoc 1567  
registerValueCallback, YVoltage 1606  
registerValueCallback, YVSource 1637  
registerValueCallback, YWakeUpMonitor 1666  
registerValueCallback, YWakeUpSchedule 1704  
registerValueCallback, YWatchdog 1746  
registerValueCallback, YWireless 1785

Relay 1290  
remove, YFiles 543  
reset, YDisplayLayer 483  
reset, YPower 1031  
reset, YSerialPort 1409  
resetAll, YDisplay 451  
resetCounter, YAnButton 146  
resetCounter, YPwmInput 1117  
resetSleepCountDown, YWakeUpMonitor 1667  
resetStatus, YMotor 906  
resetWatchdog, YWatchdog 1747  
revertFromFlash, YModule 865  
rgbMove, YColorLed 215

## S

save3DCalibration, YRefFrame 1283  
saveSequence, YDisplay 452  
saveToFlash, YModule 866  
SelectArchitecture, YAPI 26  
selectColorPen, YDisplayLayer 484  
selectEraser, YDisplayLayer 485  
selectFont, YDisplayLayer 486  
selectGrayPen, YDisplayLayer 487  
Sensor 1326  
Sequence 335, 345, 357  
SerialPort 1365  
Servo 1422  
set\_adminPassword, YNetwork 957  
set\_allSettings, YModule 867  
set\_analogCalibration, YAnButton 147  
set\_autoStart, YDataLogger 328  
set\_autoStart, YWatchdog 1748  
set\_beacon, YModule 868  
set\_beaconDriven, YDataLogger 329  
set\_bearing, YRefFrame 1284  
set\_bitDirection, YDigitalIO 403  
set\_bitOpenDrain, YDigitalIO 404  
set\_bitPolarity, YDigitalIO 405  
set\_bitState, YDigitalIO 406  
set\_blinking, YLed 733  
set\_brakingForce, YMotor 907  
set\_brightness, YDisplay 453  
set\_calibrationMax, YAnButton 148  
set\_calibrationMin, YAnButton 149  
set\_callbackCredentials, YNetwork 958  
set\_callbackEncoding, YNetwork 959  
set\_callbackMaxDelay, YNetwork 960  
set\_callbackMethod, YNetwork 961  
set\_callbackMinDelay, YNetwork 962  
set\_callbackUrl, YNetwork 963  
set\_currentValue, YAltitude 106  
set\_cutOffVoltage, YMotor 908  
set\_discoverable, YNetwork 964  
set\_drivingForce, YMotor 909  
set\_dutyCycle, YPwmOutput 1155  
set\_dutyCycleAtPowerOn, YPwmOutput 1156  
set\_enabled, YDisplay 454  
set\_enabled, YHubPort 669  
set\_enabled, YPwmOutput 1157  
set\_enabled, YServo 1448  
set\_enabledAtPowerOn, YPwmOutput 1158  
set\_enabledAtPowerOn, YServo 1449  
set\_failSafeTimeout, YMotor 910  
set\_frequency, YMotor 911  
set\_frequency, YPwmOutput 1159  
set\_highestValue, YAccelerometer 66  
set\_highestValue, YAltitude 107  
set\_highestValue, YCarbonDioxide 185  
set\_highestValue, YCompass 254  
set\_highestValue, YCurrent 293  
set\_highestValue, YGenericSensor 584  
set\_highestValue, YGyro 640  
set\_highestValue, YHumidity 704  
set\_highestValue, YLightSensor 772  
set\_highestValue, YMagnetometer 815  
set\_highestValue, YPower 1032  
set\_highestValue, YPressure 1071  
set\_highestValue, YPwmInput 1118  
set\_highestValue, YQt 1219  
set\_highestValue, YSensor 1357  
set\_highestValue, YTemperature 1489  
set\_highestValue, YTilt 1529  
set\_highestValue, YVoc 1568  
set\_highestValue, YVoltage 1607  
set\_hours, YWakeUpSchedule 1705  
set\_hslColor, YColorLed 216  
set\_logFrequency, YAccelerometer 67  
set\_logFrequency, YAltitude 108  
set\_logFrequency, YCarbonDioxide 186  
set\_logFrequency, YCompass 255  
set\_logFrequency, YCurrent 294  
set\_logFrequency, YGenericSensor 585  
set\_logFrequency, YGyro 641  
set\_logFrequency, YHumidity 705  
set\_logFrequency, YLightSensor 773  
set\_logFrequency, YMagnetometer 816  
set\_logFrequency, YPower 1033  
set\_logFrequency, YPressure 1072  
set\_logFrequency, YPwmInput 1119  
set\_logFrequency, YQt 1220  
set\_logFrequency, YSensor 1358  
set\_logFrequency, YTemperature 1490  
set\_logFrequency, YTilt 1530  
set\_logFrequency, YVoc 1569  
set\_logFrequency, YVoltage 1608  
set\_logicalName, YAccelerometer 68  
set\_logicalName, YAltitude 109  
set\_logicalName, YAnButton 150  
set\_logicalName, YCarbonDioxide 187  
set\_logicalName, YColorLed 217  
set\_logicalName, YCompass 256  
set\_logicalName, YCurrent 295  
set\_logicalName, YDataLogger 330  
set\_logicalName, YDigitalIO 407  
set\_logicalName, YDisplay 455  
set\_logicalName, YDualPower 516  
set\_logicalName, YFiles 544  
set\_logicalName, YGenericSensor 586

set\_logicalName, YGyro 642  
set\_logicalName, YHubPort 670  
set\_logicalName, YHumidity 706  
set\_logicalName, YLed 734  
set\_logicalName, YLightSensor 774  
set\_logicalName, YMagnetometer 817  
set\_logicalName, YModule 869  
set\_logicalName, YMotor 912  
set\_logicalName, YNetwork 965  
set\_logicalName, YOsControl 993  
set\_logicalName, YPower 1034  
set\_logicalName, YPressure 1073  
set\_logicalName, YPwmInput 1120  
set\_logicalName, YPwmOutput 1160  
set\_logicalName, YPwmPowerSource 1184  
set\_logicalName, YQt 1221  
set\_logicalName, YRealTimeClock 1249  
set\_logicalName, YRefFrame 1285  
set\_logicalName, YRelay 1318  
set\_logicalName, YSensor 1359  
set\_logicalName, YSerialPort 1411  
set\_logicalName, YServo 1450  
set\_logicalName, YTemperature 1491  
set\_logicalName, YTilt 1531  
set\_logicalName, YVoc 1570  
set\_logicalName, YVoltage 1609  
set\_logicalName, YVSource 1638  
set\_logicalName, YWakeUpMonitor 1668  
set\_logicalName, YWakeUpSchedule 1706  
set\_logicalName, YWatchdog 1749  
set\_logicalName, YWireless 1786  
set\_lowestValue, YAccelerometer 69  
set\_lowestValue, YAltitude 110  
set\_lowestValue, YCarbonDioxide 188  
set\_lowestValue, YCompass 257  
set\_lowestValue, YCurrent 296  
set\_lowestValue, YGenericSensor 587  
set\_lowestValue, YGyro 643  
set\_lowestValue, YHumidity 707  
set\_lowestValue, YLightSensor 775  
set\_lowestValue, YMagnetometer 818  
set\_lowestValue, YPower 1035  
set\_lowestValue, YPressure 1074  
set\_lowestValue, YPwmInput 1121  
set\_lowestValue, YQt 1222  
set\_lowestValue, YSensor 1360  
set\_lowestValue, YTemperature 1492  
set\_lowestValue, YTilt 1532  
set\_lowestValue, YVoc 1571  
set\_lowestValue, YVoltage 1610  
set\_luminosity, YLed 735  
set\_luminosity, YModule 870  
set\_maxTimeOnStateA, YRelay 1319  
set\_maxTimeOnStateA, YWatchdog 1750  
set\_maxTimeOnStateB, YRelay 1320  
set\_maxTimeOnStateB, YWatchdog 1751  
set\_measureType, YLightSensor 776  
set\_minutes, YWakeUpSchedule 1707  
set\_minutesA, YWakeUpSchedule 1708  
set\_minutesB, YWakeUpSchedule 1709  
set\_monthDays, YWakeUpSchedule 1710  
set\_months, YWakeUpSchedule 1711  
set\_mountPosition, YRefFrame 1286  
set\_neutral, YServo 1451  
set\_nextWakeUp, YWakeUpMonitor 1669  
set\_orientation, YDisplay 456  
set\_output, YRelay 1321  
set\_output, YWatchdog 1752  
set\_outputVoltage, YDigitalIO 408  
set\_overCurrentLimit, YMotor 913  
set\_period, YPwmOutput 1161  
set\_portDirection, YDigitalIO 409  
set\_portOpenDrain, YDigitalIO 410  
set\_portPolarity, YDigitalIO 411  
set\_portState, YDigitalIO 412  
set\_position, YServo 1452  
set\_positionAtPowerOn, YServo 1453  
set\_power, YLed 736  
set\_powerControl, YDualPower 517  
set\_powerDuration, YWakeUpMonitor 1670  
set\_powerMode, YPwmPowerSource 1185  
set\_primaryDNS, YNetwork 966  
set\_protocol, YSerialPort 1412  
set\_pulseDuration, YPwmOutput 1162  
set\_pwmReportMode, YPwmInput 1122  
set\_qnh, YAltitude 111  
set\_range, YServo 1454  
set\_recording, YDataLogger 331  
set\_reportFrequency, YAccelerometer 70  
set\_reportFrequency, YAltitude 112  
set\_reportFrequency, YCarbonDioxide 189  
set\_reportFrequency, YCompass 258  
set\_reportFrequency, YCurrent 297  
set\_reportFrequency, YGenericSensor 588  
set\_reportFrequency, YGyro 644  
set\_reportFrequency, YHumidity 708  
set\_reportFrequency, YLightSensor 777  
set\_reportFrequency, YMagnetometer 819  
set\_reportFrequency, YPower 1036  
set\_reportFrequency, YPressure 1075  
set\_reportFrequency, YPwmInput 1123  
set\_reportFrequency, YQt 1223  
set\_reportFrequency, YSensor 1361  
set\_reportFrequency, YTemperature 1493  
set\_reportFrequency, YTilt 1533  
set\_reportFrequency, YVoc 1572  
set\_reportFrequency, YVoltage 1611  
set\_resolution, YAccelerometer 71  
set\_resolution, YAltitude 113  
set\_resolution, YCarbonDioxide 190  
set\_resolution, YCompass 259  
set\_resolution, YCurrent 298  
set\_resolution, YGenericSensor 589  
set\_resolution, YGyro 645  
set\_resolution, YHumidity 709  
set\_resolution, YLightSensor 778  
set\_resolution, YMagnetometer 820  
set\_resolution, YPower 1037



set\_resolution, YPressure 1076  
set\_resolution, YPwmInput 1124  
set\_resolution, YQt 1224  
set\_resolution, YSensor 1362  
set\_resolution, YTemperature 1494  
set\_resolution, YTilt 1534  
set\_resolution, YVoc 1573  
set\_resolution, YVoltage 1612  
set\_rgbColor, YColorLed 218  
set\_rgbColorAtPowerOn, YColorLed 219  
set\_RTS, YSerialPort 1410  
set\_running, YWatchdog 1753  
set\_secondaryDNS, YNetwork 967  
set\_sensitivity, YAnButton 151  
set\_sensorType, YTemperature 1495  
set\_serialMode, YSerialPort 1413  
set\_signalBias, YGenericSensor 590  
set\_signalRange, YGenericSensor 591  
set\_sleepCountdown, YWakeUpMonitor 1671  
set\_starterTime, YMotor 914  
set\_startupSeq, YDisplay 457  
set\_state, YRelay 1322  
set\_state, YWatchdog 1754  
set\_stateAtPowerOn, YRelay 1323  
set\_stateAtPowerOn, YWatchdog 1755  
set\_timeUTC, YDataLogger 332  
set\_triggerDelay, YWatchdog 1756  
set\_triggerDuration, YWatchdog 1757  
set\_unit, YGenericSensor 592  
set\_unixTime, YRealTimeClock 1250  
set\_userData, YAccelerometer 72  
set\_userData, YAltitude 114  
set\_userData, YAnButton 152  
set\_userData, YCarbonDioxide 191  
set\_userData, YColorLed 220  
set\_userData, YCompass 260  
set\_userData, YCurrent 299  
set\_userData, YDataLogger 333  
set\_userData, YDigitalIO 413  
set\_userData, YDisplay 458  
set\_userData, YDualPower 518  
set\_userData, YFiles 545  
set\_userData, YGenericSensor 593  
set\_userData, YGyro 646  
set\_userData, YHubPort 671  
set\_userData, YHumidity 710  
set\_userData, YLed 737  
set\_userData, YLightSensor 779  
set\_userData, YMagnetometer 821  
set\_userData, YModule 871  
set\_userData, YMotor 915  
set\_userData, YNetwork 968  
set\_userData, YOsControl 994  
set\_userData, YPower 1038  
set\_userData, YPressure 1077  
set\_userData, YPwmInput 1125  
set\_userData, YPwmOutput 1163  
set\_userData, YPwmPowerSource 1186  
set\_userData, YQt 1225

set\_userData, YRealTimeClock 1251  
set\_userData, YRefFrame 1287  
set\_userData, YRelay 1324  
set\_userData, YSensor 1363  
set\_userData, YSerialPort 1414  
set\_userData, YServo 1455  
set\_userData, YTemperature 1496  
set\_userData, YTilt 1535  
set\_userData, YVoc 1574  
set\_userData, YVoltage 1613  
set\_userData, YVSource 1639  
set\_userData, YWakeUpMonitor 1672  
set\_userData, YWakeUpSchedule 1712  
set\_userData, YWatchdog 1758  
set\_userData, YWireless 1787  
set\_userPassword, YNetwork 969  
set\_userVar, YModule 872  
set\_utcOffset, YRealTimeClock 1252  
set\_valueInterval, YDataRun 343  
set\_valueRange, YGenericSensor 594  
set\_voltage, YVSource 1640  
set\_weekDays, YWakeUpSchedule 1713  
set\_wwwWatchdogDelay, YNetwork 970  
setAntialiasingMode, YDisplayLayer 488  
setConsoleBackground, YDisplayLayer 489  
setConsoleMargins, YDisplayLayer 490  
setConsoleWordWrap, YDisplayLayer 491  
setLayerPosition, YDisplayLayer 492  
shutdown, YOsControl 995  
Sleep, YAPI 27  
sleep, YWakeUpMonitor 1673  
sleepFor, YWakeUpMonitor 1674  
sleepUntil, YWakeUpMonitor 1675  
softAPNetwork, YWireless 1788  
Source 3, 1615  
start3DCalibration, YRefFrame 1288  
stopSequence, YDisplay 459  
Supply 495  
swapLayerContent, YDisplay 460

## T

Temperature 1457  
Tilt 1498  
Time 1227  
toggle\_bitState, YDigitalIO 414  
triggerFirmwareUpdate, YModule 873  
TriggerHubDiscovery, YAPI 28

## U

Unformatted 357  
unhide, YDisplayLayer 493  
UnregisterHub, YAPI 29  
UpdateDeviceList, YAPI 30  
updateFirmware, YModule 874  
upload, YDisplay 461  
upload, YFiles 546  
useDHCP, YNetwork 971  
useStaticIP, YNetwork 972

## V

Value 823  
Voltage 1576, 1615  
voltageMove, YVSource 1641

## W

wakeUp, YWakeUpMonitor 1676  
WakeUpMonitor 1643  
WakeUpSchedule 1678  
Watchdog 1715  
Wireless 1760  
writeArray, YSerialPort 1415  
writeBin, YSerialPort 1416  
writeHex, YSerialPort 1417  
writeLine, YSerialPort 1418  
writeMODBUS, YSerialPort 1419  
writeStr, YSerialPort 1420

## Y

YAccelerometer 34-72  
YAltitude 76-114  
YAnButton 118-152  
YAPI 12-30  
YCarbonDioxide 156-191  
yCheckLogicalName 12  
YColorLed 194-220  
YCompass 224-260  
YCurrent 264-299  
YDataLogger 303-333  
YDataRun 335-343  
YDataSet 346-355  
YDataStream 358-370  
YDigitalIO 374-414  
yDisableExceptions 13  
YDisplay 418-461  
YDisplayLayer 464-493  
YDualPower 496-518  
yEnableExceptions 14  
YFiles 521-546  
yFindAccelerometer 34  
yFindAltitude 76  
yFindAnButton 118  
yFindCarbonDioxide 156  
yFindColorLed 194  
yFindCompass 224  
yFindCurrent 264  
yFindDataLogger 303  
yFindDigitalIO 374  
yFindDisplay 418  
yFindDualPower 496  
yFindFiles 521  
yFindGenericSensor 550  
yFindGyro 599  
yFindHubPort 649  
yFindHumidity 675  
yFindLed 713  
yFindLightSensor 741

yFindMagnetometer 783  
yFindModule 831  
yFindMotor 878  
yFindNetwork 920  
yFindOsControl 975  
yFindPower 999  
yFindPressure 1042  
yFindPwmInput 1081  
yFindPwmOutput 1129  
yFindPwmPowerSource 1166  
yFindQt 1190  
yFindRealTimeClock 1228  
yFindRefFrame 1256  
yFindRelay 1292  
yFindSensor 1328  
yFindSerialPort 1368  
yFindServo 1424  
yFindTemperature 1459  
yFindTilt 1500  
yFindVoc 1539  
yFindVoltage 1578  
yFindVSource 1616  
yFindWakeUpMonitor 1645  
yFindWakeUpSchedule 1680  
yFindWatchdog 1717  
yFindWireless 1761  
yFirstAccelerometer 35  
yFirstAltitude 77  
yFirstAnButton 119  
yFirstCarbonDioxide 157  
yFirstColorLed 195  
yFirstCompass 225  
yFirstCurrent 265  
yFirstDataLogger 304  
yFirstDigitalIO 375  
yFirstDisplay 419  
yFirstDualPower 497  
yFirstFiles 522  
yFirstGenericSensor 551  
yFirstGyro 600  
yFirstHubPort 650  
yFirstHumidity 676  
yFirstLed 714  
yFirstLightSensor 742  
yFirstMagnetometer 784  
yFirstModule 832  
yFirstMotor 879  
yFirstNetwork 921  
yFirstOsControl 976  
yFirstPower 1000  
yFirstPressure 1043  
yFirstPwmInput 1082  
yFirstPwmOutput 1130  
yFirstPwmPowerSource 1167  
yFirstQt 1191  
yFirstRealTimeClock 1229  
yFirstRefFrame 1257  
yFirstRelay 1293  
yFirstSensor 1329

yFirstSerialPort 1369  
yFirstServo 1425  
yFirstTemperature 1460  
yFirstTilt 1501  
yFirstVoc 1540  
yFirstVoltage 1579  
yFirstVSource 1617  
yFirstWakeUpMonitor 1646  
yFirstWakeUpSchedule 1681  
yFirstWatchdog 1718  
yFirstWireless 1762  
yFreeAPI 15  
YGenericSensor 550-595  
yGetAPIVersion 16  
yGetTickCount 17  
YGyro 599-646  
yHandleEvents 18  
YHubPort 649-671  
YHumidity 675-710  
yInitAPI 19  
YLed 713-737  
YLightSensor 741-779  
YMagnetometer 783-821  
YMeasure 823-827  
YModule 831-874  
YMotor 878-915  
YNetwork 920-972  
Yocto-Demo 3  
Yocto-hub 648  
YOsControl 975-995  
YPower 999-1038  
yPreregisterHub 20  
YPressure 1042-1077

YPwmInput 1081-1125  
YPwmOutput 1129-1163  
YPwmPowerSource 1166-1186  
YQt 1190-1225  
YRealTimeClock 1228-1252  
YRefFrame 1256-1288  
yRegisterDeviceArrivalCallback 21  
yRegisterDeviceRemovalCallback 22  
yRegisterHub 23  
yRegisterHubDiscoveryCallback 24  
yRegisterLogFunction 25  
YRelay 1292-1324  
ySelectArchitecture 26  
YSensor 1328-1363  
YSerialPort 1368-1420  
YServo 1424-1455  
ySleep 27  
YTemperature 1459-1496  
YTilt 1500-1535  
yTriggerHubDiscovery 28  
yUnregisterHub 29  
yUpdateDeviceList 30  
YVoc 1539-1574  
YVoltage 1578-1613  
YVSource 1616-1641  
YWakeUpMonitor 1645-1676  
YWakeUpSchedule 1680-1713  
YWatchdog 1717-1758  
YWireless 1761-1788

## **Z**

zeroAdjust, YGenericSensor 595