JavaScript API Reference

Table of contents

1. Introduction

This manual is intended to be used as a reference for Yoctopuce JavaScript 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 beeing 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 Yocto-Demo with Javascript

Javascript is probably not the first language that comes to mind to control hardware, but its ease of use is a great advantage: with Javascript, you only need a text editor and a web browser to realize your first tests.

At the time of writing, the Javascript library functions with any recent browser ... except Opera. It is likely that Opera will end up working with the Yoctopuce library one of these days[1], but it is not the case right now.

Javascript is one of those languages which do not allow you to directly access the hardware layers of your computer. Therefore you need to run the Yoctopuce TCP/IP to USB gateway, named *VirtualHub*, on the machine on which your modules are connected.

2.1. Getting ready

Go to the Yoctopuce web site and download the following items:

- The Javascript programming library[2]
- The VirtualHub software[3] for Windows, Mac OS X or Linux, depending on your OS

Decompress the library files in a folder of your choice, connect your modules, run the VirtualHub software, and you are ready to start your first tests. You do not need to install any driver.

2.2. Control of the Led function

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

```
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
<SCRIPT type="text/javascript" src="yocto_led.js"></SCRIPT>

// Get access to your device, through the VirtualHub running locally
yRegisterHub('http://127.0.0.1:4444/');
var led = yFindLed("YCTOPOC1-123456.led");

// Check that the module is online to handle hot-plug
if(led.isOnline())
```

[1] Actually, as soon as Opera implements support for the HTTP Access-Control-Allow-Origin header.
[2] www.yoctopuce.com/EN/libraries.php
[3] www.yoctopuce.com/EN/virtualhub.php

```
{
    // Use led.set_power(), ...
}
```

Let us look at these lines in more details.

yocto_api.js and yocto_led.js

These two Javascript includes provide access to functions allowing you to manage Yoctopuce modules. `yocto_api.js` must always be included, `yocto_led.js` is necessary to manage modules containing a led, such as Yocto-Demo.

yRegisterHub

The `yRegisterHub` function allows you to indicate on which machine the Yoctopuce modules are located, more precisely on which machine the VirtualHub software is running. In our case, the `127.0.0.1:4444` address indicates the local machine, port `4444` (the standard port used by Yoctopuce). You can very well modify this address, and enter the address of another machine on which the VirtualHub software is running.

yFindLed

The `yFindLed` 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 also use logical names, 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.

```
var led = yFindLed("YCTOPOC1-123456.led");
var led = yFindLed("YCTOPOC1-123456.MyFunction");
var led = yFindLed("MyModule.led");
var led = yFindLed("MyModule.MyFunction");
var led = yFindLed("MyFunction");
```

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

isOnline

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

set_power

The `set_power()` function of the objet returned by `yFindLed` allows you to turn on and off the led. The argument is Y_POWER_ON or Y_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

Open your preferred text editor[4], copy the code sample below, save it in the same directory as the Yoctopuce library files and then use your preferred web browser to access this page. The code is also 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.

The example is coded to be used either from a web server, or directly by opening the file on the local machine. Note that this latest solution does not work with some versions of Internet Explorer, in particular IE 9 on Windows 7, which is not able to open network connections when working on a local file. In order to use Internet Explorer, you should load the example from a web server. No such problem exists with Chrome, Firefox or Safari.

[4] If you do not have a text editor, use Notepad rather than Microsoft Word.

If your Yocto-Demo is not connected on the host running the browser, replace in the example the address `127.0.0.1` by the IP address of the host on which the Yocto-Demo is connected and where you run the VirtualHub.

```
<HTML>
<HEAD>
 <TITLE>Hello World</TITLE>
 <SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
 <SCRIPT type="text/javascript" src="yocto_led.js"></SCRIPT>
 <SCRIPT language='javascript1.5' type='text/JavaScript'>
 <!--
 // Use explicit error handling rather than exceptions
 yDisableExceptions();

 // Setup the API to use the VirtualHub on local machine
 if(yRegisterHub('http://127.0.0.1:4444/') != YAPI_SUCCESS) {
     alert("Cannot contact VirtualHub on 127.0.0.1");
 }

 var led;

 function refresh()
 {
     var serial = document.getElementById('serial').value;
     if(serial == '') {
         // Detect any conected module suitable for the demo
         led = yFirstLed();
         if(led) {
             serial = led.module().get_serialNumber();
             document.getElementById('serial').value = serial;
         }
     }

     led = yFindLed(serial+".led");
     if(led.isOnline()) {
         document.getElementById('msg').value = '';
     } else {
         document.getElementById('msg').value = 'Module not connected';
     }
     setTimeout('refresh()',500);
 }

 function switchIt(state)
 {
     if (state) led.set_power(Y_POWER_ON);
           else led.set_power(Y_POWER_OFF);
 }
 -->
 </SCRIPT>
</HEAD>
<BODY onload='refresh();'>
 Module to use: <input id='serial'>
 <input id='msg' style='color:red;border:none;' readonly><br>
 <a href='javascript:switchIt(true);'>ON</a><br>
 <a href='javascript:switchIt(false);'>OFF</a>
</BODY>
</HTML>
```

2.3. 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.

```
<HTML>
<HEAD>
 <TITLE>Module Control</TITLE>
 <SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
 <SCRIPT language='javascript1.5'  type='text/JavaScript'>
 <!--
 // Use explicit error handling rather than exceptions
 yDisableExceptions();
 // Setup the API to use the VirtualHub on local machine
 if(yRegisterHub('http://127.0.0.1:4444/') != YAPI_SUCCESS) {
     alert("Cannot contact VirtualHub on 127.0.0.1");
 }

 var module;

 function refresh()
 {
     var serial = document.getElementById('serial').value;
     if(serial == '') {
         // Detect any conected module suitable for the demo
         module = yFirstModule().nextModule();
         if(module) {
             serial = module.get_serialNumber();
             document.getElementById('serial').value = serial;
         }
     }

     module = yFindModule(serial);
     if(module.isOnline()) {
         document.getElementById('msg').value = '';
         var html = 'serial: '+module.get_serialNumber()+'<br>';
         html += 'logical name: '+module.get_logicalName()+'<br>';
         html += 'luminosity:'+module.get_luminosity()+'%<br>';
         html += 'beacon:';
         if (module.get_beacon()==Y_BEACON_ON)
             html+="ON <a href='javascript:beacon(Y_BEACON_OFF)'>switch off</a><br>";
         else
             html+="OFF <a href='javascript:beacon(Y_BEACON_ON)'>switch on</a><br>";

         html += 'upTime: '+parseInt(module.get_upTime()/1000)+' sec<br>';
         html += 'USB current: '+module.get_usbCurrent()+' mA<br>';
         html += 'logs:<br><pre>'+module.get_lastLogs()+'</pre><br>';
         document.getElementById('data').innerHTML = html;
     } else {
         document.getElementById('msg').value = 'Module not connected';
     }
     setTimeout('refresh()',1000);
 }

 function beacon(state)
 {
     module.set_beacon(state);
     refresh();
 }
 -->
 </SCRIPT>
</HEAD>
<BODY onload='refresh();'>
 Module to use: <input id='serial'>
 <input id='msg' style='color:red;border:none;' readonly><br>
 <span id='data'></span>
</BODY>
</HTML>
```

Each property `xxx` of the module can be read thanks to a method of type `get_xxxx()`, and properties which are not read-only can be modified with the help of the `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 `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 `saveToFlash()` method. Inversely, it is possible to force the module to forget its current settings by using the `revertFromFlash()` method. The short example below allows you to modify the logical name of a module.

```
<HTML>
<HEAD>
 <TITLE>Change module settings</TITLE>
 <SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
 <SCRIPT language='javascript1.5'  type='text/JavaScript'>
 <!--
 // Use explicit error handling rather than exceptions
 yDisableExceptions();

 // Setup the API to use the VirtualHub on local machine
 if(yRegisterHub('http://127.0.0.1:4444/') != YAPI_SUCCESS) {
     alert("Cannot contact VirtualHub on 127.0.0.1");
 }

 var module;

 function refresh()
 {
     var serial = document.getElementById('serial').value;
     if(serial == '') {
         // Detect any conected module suitable for the demo
         module = yFirstModule().nextModule();
         if(module) {
             serial = module.get_serialNumber();
             document.getElementById('serial').value = serial;
         }
     }

     module = yFindModule(serial);
     if(module.isOnline()) {
         document.getElementById('msg').value = '';
         document.getElementById('curName').value = module.get_logicalName();
     } else {
         document.getElementById('msg').value = 'Module not connected';
     }
     setTimeout('refresh()',1000);
 }

 function save()
 {
     var newname = document.getElementById('newName').value;
     if (!yCheckLogicalName(newname)) {
         alert('invalid logical name');
         return;
     }
     module.set_logicalName(newname);
     module.saveToFlash();
 }
 -->
 </SCRIPT>
</HEAD>
<BODY onload='refresh();'>
 Module to use: <input id='serial'>
 <input id='msg' style='color:red;border:none;' readonly><br>
 Current name: <input id='curName' readonly><br>
 New logical name: <input id='newName'>
 <a href='javascript:save();'>Save</a>
</BODY>
</HTML>
```

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 `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 `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.

```
<HTML>
<HEAD>
 <TITLE>Modules inventory</TITLE>
 <SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
 <SCRIPT language='javascript1.5'  type='text/JavaScript'>
 <!--
 // Use explicit error handling rather than exceptions
 yDisableExceptions();

 // Setup the API to use the VirtualHub on local machine
 if(yRegisterHub('http://127.0.0.1:4444/') != YAPI_SUCCESS) {
     alert("Cannot contact VirtualHub on 127.0.0.1");
 }

 function refresh()
 {
     yUpdateDeviceList();

     var htmlcode = '';
     var module = yFirstModule();
     while(module) {
         htmlcode += module.get_serialNumber()
                     +'('+module.get_productName()+")<br>";
         module = module.nextModule();
     }
     document.getElementById('list').innerHTML=htmlcode;
     setTimeout('refresh()',500);
 }
 -->
 </SCRIPT>
</HEAD>
<BODY onload='refresh();'>
 <H1>Device list</H1>
 <tt><span id='list'></span></tt>
</BODY>
</HTML>
```

2.4. Error handling

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

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

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

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

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

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

3. Reference

3.1. General functions

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

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

`js`	`<script type='text/javascript' src='yocto_api.js'></script>`

`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
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**)

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()yCheckLogicalName() YAPI

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

```
function yCheckLogicalName( 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() yDisableExceptions()yDisableExceptions() YAPI

Disables the use of exceptions to report runtime errors.

```
function yDisableExceptions( )
```

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() yEnableExceptions()yEnableExceptions() YAPI

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

```
function yEnableExceptions( )
```

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

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

Frees dynamically allocated memory blocks used by the Yoctopuce library.

```
function yFreeAPI( )
```

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()yGetAPIVersion()

YAPI

Returns the version identifier for the Yoctopuce library in use.

```
function yGetAPIVersion( )
```

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

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

Returns :

a character string describing the library version.

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

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

```
function yGetTickCount( )
```

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()yHandleEvents() **YAPI**

Maintains the device-to-library communication channel.

```
function yHandleEvents( errmsg)
```

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

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

Parameters :

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

Returns :

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

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

Initializes the Yoctopuce programming library explicitly.

function **yInitAPI**(**mode**, **errmsg**)

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()yPreregisterHub() YAPI

Fault-tolerant alternative to RegisterHub().

```
function yPreregisterHub( url, errmsg)
```

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() yRegisterDeviceArrivalCallback() YAPI

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

```
function yRegisterDeviceArrivalCallback( 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() yRegisterDeviceRemovalCallback() YAPI

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

```
function yRegisterDeviceRemovalCallback( 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()yRegisterHub() YAPI

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

```
function yRegisterHub( url, errmsg)
```

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 a 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 you local computer, use the IP address 127.0.0.1.

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

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

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

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

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.SetTimeout() ySetTimeout()ySetTimeout() YAPI

Invoke the specified callback function after a given timeout.

```
function ySetTimeout( callback, ms_timeout, arguments)
```

This function behaves more or less like Javascript `setTimeout`, but during the waiting time, it will call `yHandleEvents` and `yUpdateDeviceList` periodically, in order to keep the API up-to-date with current devices.

Parameters :

callback	the function to call after the timeout occurs. On Microsoft Internet Explorer, the callback must be provided as a string to be evaluated.
ms_timeout	an integer corresponding to the duration of the timeout, in milliseconds.
arguments	additional arguments to be passed to the callback function can be provided, if needed (not supported on Microsoft Internet Explorer).

Returns :

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

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

Pauses the execution flow for a specified duration.

```
function ySleep( ms_duration, errmsg)
```

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.UnregisterHub() yUnregisterHub()yUnregisterHub() — YAPI

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

```
function yUnregisterHub( url)
```

Parameters :

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

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

Triggers a (re)detection of connected Yoctopuce modules.

function **yUpdateDeviceList**(**errmsg**)

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.

YAPI.UpdateDeviceList_async() yUpdateDeviceList_async() yUpdateDeviceList_async() YAPI

Triggers a (re)detection of connected Yoctopuce modules.

```
function yUpdateDeviceList_async( callback, context)
```

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.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the result code (`YAPI_SUCCESS` if the operation completes successfully) and the error message.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

3.2. Accelerometer function interface

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

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

js	`<script type='text/javascript' src='yocto_accelerometer.js'></script>`

nodejs	var yoctolib = require('yoctolib');
	var YAccelerometer = yoctolib.YAccelerometer;

php	require_once('yocto_accelerometer.php');
cpp	#include "yocto_accelerometer.h"
m	#import "yocto_accelerometer.h"
pas	uses yocto_accelerometer;
vb	yocto_accelerometer.vb
cs	yocto_accelerometer.cs
java	import com.yoctopuce.YoctoAPI.YAccelerometer;
py	from yocto_accelerometer import *

Global functions

yFindAccelerometer(func)
Retrieves an accelerometer for a given identifier.

yFirstAccelerometer()
Starts the enumeration of accelerometers currently accessible.

`YAccelerometer` methods

accelerometer→calibrateFromPoints(rawValues, refValues)
Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

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

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

accelerometer→get_currentRawValue()
Returns the uncalibrated, unrounded raw value returned by the sensor.

accelerometer→get_currentValue()
Returns the current value of the acceleration.

accelerometer→get_errorMessage()
Returns the error message of the latest error with the accelerometer.

accelerometer→get_errorType()
Returns the numerical error code of the latest error with the accelerometer.

accelerometer→get_friendlyName()
Returns a global identifier of the accelerometer in the format `MODULE_NAME.FUNCTION_NAME`.

accelerometer→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

accelerometer→get_functionId()
Returns the hardware identifier of the accelerometer, without reference to the module.

accelerometer→get_hardwareId()
Returns the unique hardware identifier of the accelerometer in the form `SERIAL.FUNCTIONID`.

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.

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()yFindAccelerometer() — YAccelerometer

Retrieves an accelerometer for a given identifier.

```
function yFindAccelerometer( 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()yFirstAccelerometer() YAccelerometer

Starts the enumeration of accelerometers currently accessible.

```
function yFirstAccelerometer( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

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

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→describe()accelerometer.describe() YAccelerometer

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

```
function 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() accelerometer→advertisedValue() accelerometer.get_advertisedValue() YAccelerometer

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

function **get_advertisedValue**( )

Returns :

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

accelerometer→get_currentRawValue() accelerometer→currentRawValue() accelerometer.get_currentRawValue() YAccelerometer

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

```
function get_currentRawValue( )
```

Returns :

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

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

accelerometer→get_currentValue()
accelerometer→currentValue()
accelerometer.get_currentValue()

YAccelerometer

Returns the current value of the acceleration.

function **get_currentValue**( )

Returns :

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

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

accelerometer→get_errorMessage() accelerometer→errorMessage() accelerometer.get_errorMessage() YAccelerometer

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

```
function get_errorMessage( )
```

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

Returns :

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

accelerometer→get_errorType()
accelerometer→errorType()
accelerometer.get_errorType()

YAccelerometer

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

```
function 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 occured while using the accelerometer object

accelerometer→get_friendlyName() accelerometer→friendlyName() accelerometer.get_friendlyName() YAccelerometer

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

function **get_friendlyName**( )

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

Returns :

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

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

accelerometer→get_functionDescriptor() accelerometer→functionDescriptor() accelerometer.get_functionDescriptor() YAccelerometer

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

```
function get_functionDescriptor( )
```

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

Returns :

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

accelerometer→get_functionId() accelerometer→functionId() accelerometer.get_functionId() YAccelerometer

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

```
function get_functionId( )
```

For example `relay1`

Returns :

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

accelerometer→get_hardwareId() accelerometer→hardwareId() accelerometer.get_hardwareId() YAccelerometer

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

```
function 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.

```
function get_highestValue( )
```

Returns :

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

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

accelerometer→get_logFrequency() accelerometer→logFrequency() accelerometer.get_logFrequency() YAccelerometer

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

```
function get_logFrequency( )
```

Returns :

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

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

accelerometer→get_logicalName() accelerometer→logicalName() accelerometer.get_logicalName() — YAccelerometer

Returns the logical name of the accelerometer.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the accelerometer. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

accelerometer→get_lowestValue()
accelerometer→lowestValue()
accelerometer.get_lowestValue() YAccelerometer

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

function **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() accelerometer→module()accelerometer.get_module() YAccelerometer

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

```
function 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_module_async() accelerometer→module_async() accelerometer.get_module_async() YAccelerometer

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

accelerometer→get_recordedData() accelerometer→recordedData() accelerometer.get_recordedData() YAccelerometer

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

```
function 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()
accelerometer→reportFrequency()
accelerometer.get_reportFrequency() — YAccelerometer

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

```
function 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.

```
function 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() accelerometer→unit()accelerometer.get_unit() YAccelerometer

Returns the measuring unit for the acceleration.

```
function get_unit( )
```

Returns :

a string corresponding to the measuring unit for the acceleration

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

accelerometer→get_userData() accelerometer→userData() accelerometer.get_userData() YAccelerometer

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

```
function get_userData( )
```

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

Returns :
the object stored previously by the caller.

accelerometer→get_xValue() accelerometer→xValue()accelerometer.get_xValue() YAccelerometer

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

```
function 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() accelerometer→yValue()accelerometer.get_yValue() YAccelerometer

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

```
function 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() accelerometer→zValue()accelerometer.get_zValue() YAccelerometer

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

```
function 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.

```
function 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→isOnline_async() accelerometer.isOnline_async() YAccelerometer

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

accelerometer→load()accelerometer.load() YAccelerometer

Preloads the accelerometer cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async() accelerometer.load_async() YAccelerometer

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

accelerometer→nextAccelerometer() accelerometer.nextAccelerometer() YAccelerometer

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

```
function 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.

```
function 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.

```
function registerValueCallback( callback)
```

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

Parameters :

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

accelerometer→set_highestValue() accelerometer→setHighestValue() accelerometer.set_highestValue() YAccelerometer

Changes the recorded maximal value observed.

```
function 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.

```
function set_logFrequency( newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→set_logicalName() accelerometer→setLogicalName() accelerometer.set_logicalName() YAccelerometer

Changes the logical name of the accelerometer.

```
function set_logicalName( newval)
```

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

Parameters :

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

Returns :

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

accelerometer→set_lowestValue()
accelerometer→setLowestValue()
accelerometer.set_lowestValue() YAccelerometer

Changes the recorded minimal value observed.

```
function set_lowestValue( newval)
```

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→set_reportFrequency()
accelerometer→setReportFrequency()
accelerometer.set_reportFrequency()

YAccelerometer

Changes the timed value notification frequency for this function.

```
function 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.

```
function 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.

```
function 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

accelerometer→wait_async() accelerometer.wait_async() YAccelerometer

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.3. AnButton function interface

Yoctopuce application programming interface allows you to measure the state of a simple button as well as to read an analog potentiometer (variable resistance). This can be 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	`<script type='text/javascript' src='yocto_anbutton.js'></script>`

nodejs	`var yoctolib = require('yoctolib');`
	`var YAnButton = yoctolib.YAnButton;`

php	`require_once('yocto_anbutton.php');`
cpp	`#include "yocto_anbutton.h"`
m	`#import "yocto_anbutton.h"`
pas	`uses yocto_anbutton;`
vb	`yocto_anbutton.vb`
cs	`yocto_anbutton.cs`
java	`import com.yoctopuce.YoctoAPI.YAnButton;`
py	`from yocto_anbutton import *`

Global functions

yFindAnButton(func)

Retrieves an analog input for a given identifier.

yFirstAnButton()

Starts the enumeration of analog inputs currently accessible.

`YAnButton` methods

anbutton→describe()

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

anbutton→get_advertisedValue()

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

anbutton→get_analogCalibration()

Tells if a calibration process is currently ongoing.

anbutton→get_calibratedValue()

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

anbutton→get_calibrationMax()

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

anbutton→get_calibrationMin()

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

anbutton→get_errorMessage()

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

anbutton→get_errorType()

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

anbutton→get_friendlyName()

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

anbutton→get_functionDescriptor()

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

anbutton→get_functionId()

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

anbutton→get_hardwareId()

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

anbutton→get_isPressed()

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

anbutton→get_lastTimePressed()

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

anbutton→get_lastTimeReleased()

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

anbutton→get_logicalName()

Returns the logical name of the analog input.

anbutton→get_module()

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

anbutton→get_module_async(callback, context)

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

anbutton→get_pulseCounter()

Returns the pulse counter value

anbutton→get_pulseTimer()

Returns the timer of the pulses counter (ms)

anbutton→get_rawValue()

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

anbutton→get_sensitivity()

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

anbutton→get_userData()

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

anbutton→isOnline()

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

anbutton→isOnline_async(callback, context)

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

anbutton→load(msValidity)

Preloads the analog input cache with a specified validity duration.

anbutton→load_async(msValidity, callback, context)

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

anbutton→nextAnButton()

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

anbutton→registerValueCallback(callback)

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

anbutton→resetCounter()

Returns the pulse counter value as well as his timer

anbutton→set_analogCalibration(newval)

Starts or stops the calibration process.

anbutton→set_calibrationMax(newval)

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()yFindAnButton() — YAnButton

Retrieves an analog input for a given identifier.

```
function yFindAnButton( 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() yFirstAnButton()yFirstAnButton()

YAnButton

Starts the enumeration of analog inputs currently accessible.

```
function yFirstAnButton( )
```

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`.

```
function 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() anbutton→advertisedValue() anbutton.get_advertisedValue() YAnButton

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

```
function get_advertisedValue( )
```

Returns :

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

anbutton→get_analogCalibration() anbutton→analogCalibration() anbutton.get_analogCalibration() YAnButton

Tells if a calibration process is currently ongoing.

function **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).

```
function get_calibratedValue( )
```

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

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

anbutton→get_calibrationMax() anbutton→calibrationMax() anbutton.get_calibrationMax() YAnButton

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

```
function 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).

```
function 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.

```
function get_errorMessage( )
```

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

Returns :

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

anbutton→get_errorType() anbutton→errorType()anbutton.get_errorType() YAnButton

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

```
function 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 occured while using the analog input object

anbutton→get_friendlyName() anbutton→friendlyName() anbutton.get_friendlyName() YAnButton

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

```
function get_friendlyName( )
```

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

Returns :

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

anbutton→get_functionDescriptor() anbutton→functionDescriptor() anbutton.get_functionDescriptor() YAnButton

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

```
function 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() anbutton→functionId()anbutton.get_functionId() YAnButton

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

function **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() anbutton→hardwareId()anbutton.get_hardwareId() YAnButton

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

```
function 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() anbutton→isPressed()anbutton.get_isPressed() YAnButton

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

function **get_isPressed**( )

Returns :

either `Y_ISPRESSED_FALSE` or `Y_ISPRESSED_TRUE`, according to true if the input (considered as binary) is active (closed contact), and false otherwise

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

anbutton→get_lastTimePressed() anbutton→lastTimePressed() anbutton.get_lastTimePressed() YAnButton

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

```
function get_lastTimePressed( )
```

Returns :

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

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

anbutton→get_lastTimeReleased() anbutton→lastTimeReleased() anbutton.get_lastTimeReleased() YAnButton

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

```
function get_lastTimeReleased( )
```

Returns :

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

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

anbutton→get_logicalName() anbutton→logicalName()anbutton.get_logicalName() YAnButton

Returns the logical name of the analog input.

```
function 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() anbutton→module()anbutton.get_module() YAnButton

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

```
function 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_module_async() anbutton→module_async() anbutton.get_module_async() YAnButton

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

anbutton→get_pulseCounter()
anbutton→pulseCounter()
anbutton.get_pulseCounter()

YAnButton

Returns the pulse counter value

```
function get_pulseCounter( )
```

Returns :

an integer corresponding to the pulse counter value

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

anbutton→get_pulseTimer() anbutton→pulseTimer()anbutton.get_pulseTimer() YAnButton

Returns the timer of the pulses counter (ms)

```
function get_pulseTimer( )
```

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

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

anbutton→get_rawValue() anbutton→rawValue()anbutton.get_rawValue() YAnButton

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

```
function 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() anbutton→sensitivity()anbutton.get_sensitivity() YAnButton

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

function **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() anbutton→userData()anbutton.get_userData() YAnButton

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

```
function 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.

```
function 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→isOnline_async() anbutton.isOnline_async() YAnButton

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

function **isOnline_async**(**callback**, **context**)

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

anbutton→load()anbutton.load() YAnButton

Preloads the analog input cache with a specified validity duration.

```
function 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 trafic 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→load_async()anbutton.load_async() YAnButton

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

anbutton→nextAnButton()anbutton.nextAnButton() YAnButton

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

```
function 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.

```
function 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

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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() anbutton→setSensitivity()anbutton.set_sensitivity() YAnButton

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

```
function 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.

anbuton→set_userData() anbutton→setUserData()anbutton.set_userData() YAnButton

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

```
function 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

anbutton→wait_async()anbutton.wait_async() YAnButton

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.4. CarbonDioxide function interface

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

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

js	<script type='text/javascript' src='yocto_carbondioxide.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YCarbonDioxide = yoctolib.YCarbonDioxide;

php	require_once('yocto_carbondioxide.php');
cpp	#include "yocto_carbondioxide.h"
m	#import "yocto_carbondioxide.h"
pas	uses yocto_carbondioxide;
vb	yocto_carbondioxide.vb
cs	yocto_carbondioxide.cs
java	import com.yoctopuce.YoctoAPI.YCarbonDioxide;
py	from yocto_carbondioxide import *

Global functions

yFindCarbonDioxide(func)

Retrieves a CO2 sensor for a given identifier.

yFirstCarbonDioxide()

Starts the enumeration of CO2 sensors currently accessible.

`YCarbonDioxide` methods

carbondioxide→calibrateFromPoints(rawValues, refValues)

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

carbondioxide→describe()

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

carbondioxide→get_advertisedValue()

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

carbondioxide→get_currentRawValue()

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

carbondioxide→get_currentValue()

Returns the current value of the CO2 concentration.

carbondioxide→get_errorMessage()

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

carbondioxide→get_errorType()

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

carbondioxide→get_friendlyName()

Returns a global identifier of the CO2 sensor in the format `MODULE_NAME.FUNCTION_NAME`.

carbondioxide→get_functionDescriptor()

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

carbondioxide→get_functionId()

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

carbondioxide→get_hardwareId()

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

carbondioxide→get_highestValue()
Returns the maximal value observed for the CO2 concentration since the device was started.

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

carbondioxide→get_logicalName()
Returns the logical name of the CO2 sensor.

carbondioxide→get_lowestValue()
Returns the minimal value observed for the CO2 concentration since the device was started.

carbondioxide→get_module()
Gets the `YModule` object for the device on which the function is located.

carbondioxide→get_module_async(callback, context)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

carbondioxide→get_recordedData(startTime, endTime)
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

carbondioxide→get_reportFrequency()
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

carbondioxide→get_resolution()
Returns the resolution of the measured values.

carbondioxide→get_unit()
Returns the measuring unit for the CO2 concentration.

carbondioxide→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

carbondioxide→isOnline()
Checks if the CO2 sensor is currently reachable, without raising any error.

carbondioxide→isOnline_async(callback, context)
Checks if the CO2 sensor is currently reachable, without raising any error (asynchronous version).

carbondioxide→load(msValidity)
Preloads the CO2 sensor cache with a specified validity duration.

carbondioxide→loadCalibrationPoints(rawValues, refValues)
Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

carbondioxide→load_async(msValidity, callback, context)
Preloads the CO2 sensor cache with a specified validity duration (asynchronous version).

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

carbondioxide→registerTimedReportCallback(callback)
Registers the callback function that is invoked on every periodic timed notification.

carbondioxide→registerValueCallback(callback)
Registers the callback function that is invoked on every change of advertised value.

carbondioxide→set_highestValue(newval)
Changes the recorded maximal value observed.

carbondioxide→set_logFrequency(newval)
Changes the datalogger recording frequency for this function.

carbondioxide→set_logicalName(newval)
Changes the logical name of the CO2 sensor.

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()yFindCarbonDioxide()

YCarbonDioxide

Retrieves a CO2 sensor for a given identifier.

```
function yFindCarbonDioxide( 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()yFirstCarbonDioxide() YCarbonDioxide

Starts the enumeration of CO2 sensors currently accessible.

```
function yFirstCarbonDioxide( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

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

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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`.

```
function 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() carbondioxide→advertisedValue() carbondioxide.get_advertisedValue() YCarbonDioxide

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

```
function 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.

```
function get_currentRawValue( )
```

Returns :

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

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

carbondioxide→get_currentValue()
carbondioxide→currentValue()
carbondioxide.get_currentValue()

YCarbonDioxide

Returns the current value of the CO2 concentration.

```
function get_currentValue( )
```

Returns :

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

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

carbondioxide→get_errorMessage() carbondioxide→errorMessage() carbondioxide.get_errorMessage() YCarbonDioxide

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

```
function get_errorMessage( )
```

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

Returns :
a string corresponding to the latest error message that occured while using 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.

```
function 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 occured 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`.

```
function get_friendlyName( )
```

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

Returns :

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

carbondioxide→get_functionDescriptor()
carbondioxide→functionDescriptor()
carbondioxide.get_functionDescriptor()

YCarbonDioxide

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

```
function 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.

```
function 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`.

function **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.

```
function get_highestValue( )
```

Returns :

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

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

carbondioxide→get_logFrequency() carbondioxide→logFrequency() carbondioxide.get_logFrequency() YCarbonDioxide

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

```
function 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.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the CO2 sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

carbondioxide→get_lowestValue() carbondioxide→lowestValue() carbondioxide.get_lowestValue() YCarbonDioxide

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

```
function 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.

```
function 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_module_async() carbondioxide→module_async() carbondioxide.get_module_async()

YCarbonDioxide

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

carbondioxide→get_recordedData() carbondioxide→recordedData() carbondioxide.get_recordedData() — YCarbonDioxide

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

```
function 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.

```
function get_reportFrequency( )
```

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

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

carbondioxide→get_resolution() carbondioxide→resolution() carbondioxide.get_resolution() YCarbonDioxide

Returns the resolution of the measured values.

```
function 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() carbondioxide→unit()carbondioxide.get_unit() YCarbonDioxide

Returns the measuring unit for the CO2 concentration.

```
function get_unit( )
```

Returns :

a string corresponding to the measuring unit for the CO2 concentration

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

carbondioxide→get_userData() carbondioxide→userData() carbondioxide.get_userData()

YCarbonDioxide

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

```
function 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.

```
function 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→isOnline_async() carbondioxide.isOnline_async() YCarbonDioxide

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

carbondioxide→load()carbondioxide.load() YCarbonDioxide

Preloads the CO2 sensor cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async() carbondioxide.load_async() YCarbonDioxide

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

function **load_async**(**msValidity**, **callback**, **context**)

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

carbondioxide→nextCarbonDioxide() carbondioxide.nextCarbonDioxide() YCarbonDioxide

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

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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

carbondioxide→wait_async() carbondioxide.wait_async() YCarbonDioxide

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.5. ColorLed function interface

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

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

js	<script type='text/javascript' src='yocto_colorled.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YColorLed = yoctolib.YColorLed;

php	require_once('yocto_colorled.php');
cpp	#include "yocto_colorled.h"
m	#import "yocto_colorled.h"
pas	uses yocto_colorled;
vb	yocto_colorled.vb
cs	yocto_colorled.cs
java	import com.yoctopuce.YoctoAPI.YColorLed;
py	from yocto_colorled import *

Global functions

yFindColorLed(func)

Retrieves an RGB led for a given identifier.

yFirstColorLed()

Starts the enumeration of RGB leds currently accessible.

`YColorLed` methods

colorled→describe()

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

colorled→get_advertisedValue()

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

colorled→get_errorMessage()

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

colorled→get_errorType()

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

colorled→get_friendlyName()

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

colorled→get_functionDescriptor()

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

colorled→get_functionId()

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

colorled→get_hardwareId()

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

colorled→get_hslColor()

Returns the current HSL color of the led.

colorled→get_logicalName()

Returns the logical name of the RGB led.

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()yFindColorLed() YColorLed

Retrieves an RGB led for a given identifier.

```
function yFindColorLed( 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()yFirstColorLed() YColorLed

Starts the enumeration of RGB leds currently accessible.

```
function yFirstColorLed( )
```

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`.

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

colorled→get_advertisedValue()
colorled→advertisedValue()
colorled.get_advertisedValue() YColorLed

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

```
function get_advertisedValue( )
```

Returns :

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

colorled→get_errorMessage()
colorled→errorMessage()
colorled.get_errorMessage() — YColorLed

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

```
function get_errorMessage( )
```

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

Returns :
a string corresponding to the latest error message that occured while using the RGB led object

colorled→get_errorType() colorled→errorType()colorled.get_errorType() YColorLed

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

```
function 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 occured while using the RGB led object

colorled→get_friendlyName() colorled→friendlyName()colorled.get_friendlyName() YColorLed

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

```
function get_friendlyName( )
```

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

Returns :

a string that uniquely identifies the RGB led using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

colorled→get_functionDescriptor() colorled→functionDescriptor() colorled.get_functionDescriptor() YColorLed

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

```
function 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() colorled→functionId()colorled.get_functionId() YColorLed

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

```
function 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() colorled→hardwareId()colorled.get_hardwareId() YColorLed

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

```
function 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() colorled→hslColor()colorled.get_hslColor() YColorLed

Returns the current HSL color of the led.

```
function 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() colorled→logicalName()colorled.get_logicalName() YColorLed

Returns the logical name of the RGB led.

```
function 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() colorled→module()colorled.get_module() YColorLed

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

```
function 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_module_async() colorled→module_async() colorled.get_module_async() — YColorLed

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

colorled→get_rgbColor() colorled→rgbColor()colorled.get_rgbColor() YColorLed

Returns the current RGB color of the led.

```
function 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() colorled→rgbColorAtPowerOn() colorled.get_rgbColorAtPowerOn() YColorLed

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

function **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() colorled→userData()colorled.get_userData() YColorLed

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

```
function 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.

```
function 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.

```
function 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→isOnline_async()colorled.isOnline_async() YColorLed

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

colorled→load()colorled.load() YColorLed

Preloads the RGB led cache with a specified validity duration.

```
function 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 trafic 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→load_async()colorled.load_async() YColorLed

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

colorled→nextColorLed()colorled.nextColorLed() YColorLed

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

```
function 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.

```
function 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.

```
function 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() colorled→setHslColor()colorled.set_hslColor() YColorLed

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

```
function set_hslColor( newval)
```

Encoding is done as follows: 0xHHSSLL.

Parameters :

newval an integer corresponding to the current color of the led, using a color HSL

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

colorled→set_logicalName() colorled→setLogicalName() colorled.set_logicalName() YColorLed

Changes the logical name of the RGB led.

```
function 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() colorled→setRgbColor()colorled.set_rgbColor() YColorLed

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

```
function set_rgbColor( newval)
```

Encoding is done as follows: 0xRRGGBB.

Parameters :

newval an integer corresponding to the current color of the led, using a RGB color

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

colorled→set_rgbColorAtPowerOn()
colorled→setRgbColorAtPowerOn()
colorled.set_rgbColorAtPowerOn() YColorLed

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

```
function 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() colorled→setUserData()colorled.set_userData() YColorLed

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

```
function 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

colorled→wait_async()colorled.wait_async() YColorLed

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.6. Compass function interface

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

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

js	`<script type='text/javascript' src='yocto_compass.js'></script>`

nodejs	var yoctolib = require('yoctolib');
	var YCompass = yoctolib.YCompass;

php	require_once('yocto_compass.php');
cpp	#include "yocto_compass.h"
m	#import "yocto_compass.h"
pas	uses yocto_compass;
vb	yocto_compass.vb
cs	yocto_compass.cs
java	import com.yoctopuce.YoctoAPI.YCompass;
py	from yocto_compass import *

Global functions

yFindCompass(**func**)

Retrieves a compass for a given identifier.

yFirstCompass()

Starts the enumeration of compasses currently accessible.

`YCompass` methods

compass→calibrateFromPoints(**rawValues**, **refValues**)

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

compass→describe()

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

compass→get_advertisedValue()

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

compass→get_currentRawValue()

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

compass→get_currentValue()

Returns the current value of the relative bearing.

compass→get_errorMessage()

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

compass→get_errorType()

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

compass→get_friendlyName()

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

compass→get_functionDescriptor()

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

compass→get_functionId()

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

compass→get_hardwareId()

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

compass→get_highestValue()
Returns the maximal value observed for the relative bearing since the device was started.

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

compass→get_logicalName()
Returns the logical name of the compass.

compass→get_lowestValue()
Returns the minimal value observed for the relative bearing since the device was started.

compass→get_magneticHeading()
Returns the magnetic heading, regardless of the configured bearing.

compass→get_module()
Gets the `YModule` object for the device on which the function is located.

compass→get_module_async(callback, context)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

compass→get_recordedData(startTime, endTime)
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

compass→get_reportFrequency()
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

compass→get_resolution()
Returns the resolution of the measured values.

compass→get_unit()
Returns the measuring unit for the relative bearing.

compass→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

compass→isOnline()
Checks if the compass is currently reachable, without raising any error.

compass→isOnline_async(callback, context)
Checks if the compass is currently reachable, without raising any error (asynchronous version).

compass→load(msValidity)
Preloads the compass cache with a specified validity duration.

compass→loadCalibrationPoints(rawValues, refValues)
Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

compass→load_async(msValidity, callback, context)
Preloads the compass cache with a specified validity duration (asynchronous version).

compass→nextCompass()
Continues the enumeration of compasses started using `yFirstCompass()`.

compass→registerTimedReportCallback(callback)
Registers the callback function that is invoked on every periodic timed notification.

compass→registerValueCallback(callback)
Registers the callback function that is invoked on every change of advertised value.

compass→set_highestValue(newval)
Changes the recorded maximal value observed.

compass→set_logFrequency(newval)
Changes the datalogger recording frequency for this function.

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()yFindCompass() — YCompass

Retrieves a compass for a given identifier.

```
function yFindCompass( 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()yFirstCompass() YCompass

Starts the enumeration of compasses currently accessible.

```
function yFirstCompass( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

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

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→describe()compass.describe() YCompass

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

```
function 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() compass→advertisedValue() compass.get_advertisedValue() YCompass

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

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the compass (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

compass→get_currentRawValue() compass→currentRawValue() compass.get_currentRawValue() YCompass

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

```
function get_currentRawValue( )
```

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

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

compass→get_currentValue() compass→currentValue() compass.get_currentValue() — YCompass

Returns the current value of the relative bearing.

```
function get_currentValue( )
```

Returns :
a floating point number corresponding to the current value of the relative bearing

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

compass→get_errorMessage() compass→errorMessage() compass.get_errorMessage() YCompass

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

```
function get_errorMessage( )
```

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

Returns :
a string corresponding to the latest error message that occured while using the compass object

compass→get_errorType() compass→errorType()compass.get_errorType() YCompass

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

```
function 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 occured 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`.

```
function get_friendlyName( )
```

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

Returns :

a string that uniquely identifies the compass using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

compass→get_functionDescriptor()
compass→functionDescriptor()
compass.get_functionDescriptor() YCompass

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

```
function 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() compass→functionId()compass.get_functionId() YCompass

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

```
function 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() compass→hardwareId()compass.get_hardwareId() YCompass

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

function **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() compass→highestValue() compass.get_highestValue() YCompass

Returns the maximal value observed for the relative bearing since the device was started.

```
function get_highestValue( )
```

Returns :

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

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

compass→get_logFrequency()
compass→logFrequency()
compass.get_logFrequency()

YCompass

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

```
function 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() compass→logicalName()compass.get_logicalName() YCompass

Returns the logical name of the compass.

```
function 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() compass→lowestValue()compass.get_lowestValue() YCompass

Returns the minimal value observed for the relative bearing since the device was started.

```
function get_lowestValue( )
```

Returns :

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

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

compass→get_magneticHeading() compass→magneticHeading() compass.get_magneticHeading() YCompass

Returns the magnetic heading, regardless of the configured bearing.

```
function 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() compass→module()compass.get_module() YCompass

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

```
function 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_module_async() compass→module_async() compass.get_module_async()

YCompass

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

compass→get_recordedData() compass→recordedData() compass.get_recordedData() YCompass

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

```
function 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.

compass→get_reportFrequency()
compass→reportFrequency()
compass.get_reportFrequency()

YCompass

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

```
function 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() compass→resolution()compass.get_resolution() YCompass

Returns the resolution of the measured values.

```
function 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() compass→unit()compass.get_unit() YCompass

Returns the measuring unit for the relative bearing.

```
function 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() compass→userData()compass.get_userData() YCompass

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

```
function 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.

```
function isOnline( )
```

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

Returns :

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

compass→isOnline_async() compass.isOnline_async() YCompass

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

compass→load()compass.load() YCompass

Preloads the compass cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async()compass.load_async() YCompass

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

compass→nextCompass()compass.nextCompass() YCompass

Continues the enumeration of compasses started using `yFirstCompass()`.

```
function 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.

```
function 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.

```
function registerValueCallback( callback)
```

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

Parameters :

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

compass→set_highestValue()
compass→setHighestValue()
compass.set_highestValue()

YCompass

Changes the recorded maximal value observed.

```
function 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.

```
function set_logFrequency( newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→set_logicalName()
compass→setLogicalName()
compass.set_logicalName()

YCompass

Changes the logical name of the compass.

```
function 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.

```
function set_lowestValue( newval)
```

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→set_reportFrequency() compass→setReportFrequency() compass.set_reportFrequency() YCompass

Changes the timed value notification frequency for this function.

```
function 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() compass→setResolution()compass.set_resolution() YCompass

Changes the resolution of the measured physical values.

```
function 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() compass→setUserData()compass.set_userData() YCompass

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

```
function 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

compass→wait_async()compass.wait_async() YCompass

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.7. Current function interface

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

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

js	<script type='text/javascript' src='yocto_current.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YCurrent = yoctolib.YCurrent;

php	require_once('yocto_current.php');
cpp	#include "yocto_current.h"
m	#import "yocto_current.h"
pas	uses yocto_current;
vb	yocto_current.vb
cs	yocto_current.cs
java	import com.yoctopuce.YoctoAPI.YCurrent;
py	from yocto_current import *

Global functions

yFindCurrent(func)
Retrieves a current sensor for a given identifier.

yFirstCurrent()
Starts the enumeration of current sensors currently accessible.

`YCurrent` methods

current→calibrateFromPoints(rawValues, refValues)
Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

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

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

current→get_currentRawValue()
Returns the uncalibrated, unrounded raw value returned by the sensor.

current→get_currentValue()
Returns the current measure for the current.

current→get_errorMessage()
Returns the error message of the latest error with the current sensor.

current→get_errorType()
Returns the numerical error code of the latest error with the current sensor.

current→get_friendlyName()
Returns a global identifier of the current sensor in the format `MODULE_NAME.FUNCTION_NAME`.

current→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

current→get_functionId()
Returns the hardware identifier of the current sensor, without reference to the module.

current→get_hardwareId()
Returns the unique hardware identifier of the current sensor in the form `SERIAL.FUNCTIONID`.

current→get_highestValue()
Returns the maximal value observed for the current.

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

current→get_logicalName()
Returns the logical name of the current sensor.

current→get_lowestValue()
Returns the minimal value observed for the current.

current→get_module()
Gets the `YModule` object for the device on which the function is located.

current→get_module_async(callback, context)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

current→get_recordedData(startTime, endTime)
Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

current→get_reportFrequency()
Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

current→get_resolution()
Returns the resolution of the measured values.

current→get_unit()
Returns the measuring unit for the current.

current→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

current→isOnline()
Checks if the current sensor is currently reachable, without raising any error.

current→isOnline_async(callback, context)
Checks if the current sensor is currently reachable, without raising any error (asynchronous version).

current→load(msValidity)
Preloads the current sensor cache with a specified validity duration.

current→loadCalibrationPoints(rawValues, refValues)
Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

current→load_async(msValidity, callback, context)
Preloads the current sensor cache with a specified validity duration (asynchronous version).

current→nextCurrent()
Continues the enumeration of current sensors started using `yFirstCurrent()`.

current→registerTimedReportCallback(callback)
Registers the callback function that is invoked on every periodic timed notification.

current→registerValueCallback(callback)
Registers the callback function that is invoked on every change of advertised value.

current→set_highestValue(newval)
Changes the recorded maximal value observed pour the current.

current→set_logFrequency(newval)
Changes the datalogger recording frequency for this function.

current→set_logicalName(newval)
Changes the logical name of the current sensor.

current→set_lowestValue(newval)

Changes the recorded minimal value observed pour the current.

current→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

current→set_resolution(newval)

Changes the resolution of the measured values.

current→set_userData(data)

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

current→wait_async(callback, context)

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

YCurrent.FindCurrent() yFindCurrent()yFindCurrent() YCurrent

Retrieves a current sensor for a given identifier.

```
function yFindCurrent( 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() yFirstCurrent()yFirstCurrent() — YCurrent

Starts the enumeration of current sensors currently accessible.

```
function yFirstCurrent( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

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

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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`.

```
function 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()
current→advertisedValue()
current.get_advertisedValue()

YCurrent

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

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the current sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

current→get_currentRawValue()
current→currentRawValue()
current.get_currentRawValue()

YCurrent

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

```
function get_currentRawValue( )
```

Returns :

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

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

current→get_currentValue() current→currentValue()current.get_currentValue() YCurrent

Returns the current measure for the current.

function **get_currentValue**( )

Returns :
a floating point number corresponding to the current measure for the current

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

current→get_errorMessage() current→errorMessage()current.get_errorMessage() YCurrent

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

```
function get_errorMessage( )
```

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

Returns :
a string corresponding to the latest error message that occured while using the current sensor object

current→get_errorType() current→errorType()current.get_errorType() YCurrent

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

```
function 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 occured while using the current sensor object

current→get_friendlyName() current→friendlyName()current.get_friendlyName() YCurrent

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

```
function get_friendlyName( )
```

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

Returns :

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

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

current→get_functionDescriptor()
current→functionDescriptor()
current.get_functionDescriptor() — YCurrent

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

```
function 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() current→functionId()current.get_functionId() YCurrent

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

```
function 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() current→hardwareId()current.get_hardwareId() YCurrent

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

```
function 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() current→highestValue()current.get_highestValue() YCurrent

Returns the maximal value observed for the current.

```
function get_highestValue( )
```

Returns :

a floating point number corresponding to the maximal value observed for the current

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

current→get_logFrequency() current→logFrequency()current.get_logFrequency() YCurrent

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

```
function 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() current→logicalName()current.get_logicalName() YCurrent

Returns the logical name of the current sensor.

```
function 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() current→lowestValue()current.get_lowestValue() YCurrent

Returns the minimal value observed for the current.

```
function get_lowestValue( )
```

Returns :

a floating point number corresponding to the minimal value observed for the current

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

current→get_module() current→module()current.get_module() YCurrent

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

```
function 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_module_async() current→module_async() current.get_module_async() YCurrent

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

current→get_recordedData() current→recordedData()current.get_recordedData() YCurrent

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

```
function 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.

current→get_reportFrequency() current→reportFrequency() current.get_reportFrequency() YCurrent

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

```
function 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() current→resolution()current.get_resolution() YCurrent

Returns the resolution of the measured values.

```
function 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()
current→unit()current.get_unit() YCurrent

Returns the measuring unit for the current.

```
function 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() current→userData()current.get_userData() YCurrent

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

```
function 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.

```
function 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→isOnline_async()current.isOnline_async() YCurrent

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

current→load()current.load() YCurrent

Preloads the current sensor cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async()current.load_async() YCurrent

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

current→nextCurrent()current.nextCurrent() YCurrent

Continues the enumeration of current sensors started using `yFirstCurrent()`.

```
function 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.

```
function 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.

```
function registerValueCallback( callback)
```

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

Parameters :

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

current→set_highestValue()
current→setHighestValue()
current.set_highestValue()

YCurrent

Changes the recorded maximal value observed pour the current.

```
function set_highestValue( newval)
```

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

current→set_logFrequency() current→setLogFrequency() current.set_logFrequency() YCurrent

Changes the datalogger recording frequency for this function.

```
function 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() current→setLogicalName()current.set_logicalName() YCurrent

Changes the logical name of the current sensor.

```
function 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() current→setLowestValue()current.set_lowestValue() YCurrent

Changes the recorded minimal value observed pour the current.

```
function set_lowestValue( newval)
```

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

current→set_reportFrequency() current→setReportFrequency() current.set_reportFrequency() YCurrent

Changes the timed value notification frequency for this function.

```
function 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() current→setResolution()current.set_resolution() YCurrent

Changes the resolution of the measured values.

```
function set_resolution( newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

current→set_userData() current→setUserData()current.set_userData() YCurrent

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

```
function 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

current→wait_async()current.wait_async() YCurrent

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.8. DataLogger function interface

Yoctopuce sensors include a non-volatile memory capable of storing ongoing measured data automatically, without requiring a permanent connection to a computer. The DataLogger function controls the global parameters of the internal data logger.

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

js	<script type='text/javascript' src='yocto_datalogger.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YDataLogger = yoctolib.YDataLogger;

php	require_once('yocto_datalogger.php');
cpp	#include "yocto_datalogger.h"
m	#import "yocto_datalogger.h"
pas	uses yocto_datalogger;
vb	yocto_datalogger.vb
cs	yocto_datalogger.cs
java	import com.yoctopuce.YoctoAPI.YDataLogger;
py	from yocto_datalogger import *

Global functions

yFindDataLogger(func)

Retrieves a data logger for a given identifier.

yFirstDataLogger()

Starts the enumeration of data loggers currently accessible.

`YDataLogger` methods

datalogger→describe()

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

datalogger→forgetAllDataStreams()

Clears the data logger memory and discards all recorded data streams.

datalogger→get_advertisedValue()

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

datalogger→get_autoStart()

Returns the default activation state of the data logger on power up.

datalogger→get_currentRunIndex()

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

datalogger→get_dataSets()

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

datalogger→get_dataStreams(v)

Builds a list of all data streams hold by the data logger (legacy method).

datalogger→get_errorMessage()

Returns the error message of the latest error with the data logger.

datalogger→get_errorType()

Returns the numerical error code of the latest error with the data logger.

datalogger→get_friendlyName()

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

datalogger→get_functionDescriptor()

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

datalogger→get_functionId()

Returns the hardware identifier of the data logger, without reference to the module.

datalogger→get_hardwareId()

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

datalogger→get_logicalName()

Returns the logical name of the data logger.

datalogger→get_module()

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

datalogger→get_module_async(callback, context)

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

datalogger→get_recording()

Returns the current activation state of the data logger.

datalogger→get_timeUTC()

Returns the Unix timestamp for current UTC time, if known.

datalogger→get_userData()

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

datalogger→isOnline()

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

datalogger→isOnline_async(callback, context)

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

datalogger→load(msValidity)

Preloads the data logger cache with a specified validity duration.

datalogger→load_async(msValidity, callback, context)

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

datalogger→nextDataLogger()

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

datalogger→registerValueCallback(callback)

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

datalogger→set_autoStart(newval)

Changes the default activation state of the data logger on power up.

datalogger→set_logicalName(newval)

Changes the logical name of the data logger.

datalogger→set_recording(newval)

Changes the activation state of the data logger to start/stop recording data.

datalogger→set_timeUTC(newval)

Changes the current UTC time reference used for recorded data.

datalogger→set_userData(data)

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

datalogger→wait_async(callback, context)

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

YDataLogger.FindDataLogger() yFindDataLogger()yFindDataLogger() YDataLogger

Retrieves a data logger for a given identifier.

```
function yFindDataLogger( 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() yFirstDataLogger()yFirstDataLogger()

YDataLogger

Starts the enumeration of data loggers currently accessible.

```
function yFirstDataLogger( )
```

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`.

```
function 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 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.

```
function forgetAllDataStreams( )
```

This method also resets the current run index to zero.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

datalogger→get_advertisedValue() datalogger→advertisedValue() datalogger.get_advertisedValue() YDataLogger

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

```
function 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() datalogger→autoStart()datalogger.get_autoStart() YDataLogger

Returns the default activation state of the data logger on power up.

```
function get_autoStart( )
```

Returns :

either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the default activation state of the data logger on power up

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

datalogger→get_currentRunIndex()
datalogger→currentRunIndex()
datalogger.get_currentRunIndex()

YDataLogger

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

```
function 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() datalogger→dataSets()datalogger.get_dataSets() YDataLogger

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

```
function 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).

```
function get_dataStreams( v)
```

The caller must pass by reference an empty array to hold YDataStream objects, and the function fills it with objects describing available data sequences.

This is the old way to retrieve data from the DataLogger. For new applications, you should rather use `get_dataSets()` method, or call directly `get_recordedData()` on the sensor object.

Parameters :

v an array of YDataStream objects to be filled in

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

datalogger→get_errorMessage()
datalogger→errorMessage()
datalogger.get_errorMessage()

YDataLogger

Returns the error message of the latest error with the data logger.

```
function get_errorMessage( )
```

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

Returns :

a string corresponding to the latest error message that occured while using the data logger object

datalogger→get_errorType() datalogger→errorType()datalogger.get_errorType() YDataLogger

Returns the numerical error code of the latest error with the data logger.

```
function 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 occured while using the data logger object

datalogger→get_friendlyName()
datalogger→friendlyName()
datalogger.get_friendlyName()

YDataLogger

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

```
function get_friendlyName( )
```

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

Returns :

a string that uniquely identifies the data logger using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

datalogger→get_functionDescriptor() datalogger→functionDescriptor() datalogger.get_functionDescriptor() YDataLogger

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

```
function 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() datalogger→functionId()datalogger.get_functionId() YDataLogger

Returns the hardware identifier of the data logger, without reference to the module.

```
function 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`.

```
function 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()
datalogger→logicalName()
datalogger.get_logicalName() YDataLogger

Returns the logical name of the data logger.

```
function 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() datalogger→module()datalogger.get_module() YDataLogger

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

```
function 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_module_async() datalogger→module_async() datalogger.get_module_async() YDataLogger

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

datalogger→get_recording() datalogger→recording()datalogger.get_recording() YDataLogger

Returns the current activation state of the data logger.

```
function 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() datalogger→timeUTC()datalogger.get_timeUTC() YDataLogger

Returns the Unix timestamp for current UTC time, if known.

```
function 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() datalogger→userData()datalogger.get_userData() YDataLogger

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

```
function 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.

```
function 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→isOnline_async() datalogger.isOnline_async() YDataLogger

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

datalogger→load()datalogger.load() YDataLogger

Preloads the data logger cache with a specified validity duration.

```
function 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 trafic 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→load_async()datalogger.load_async() YDataLogger

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

datalogger→nextDataLogger() datalogger.nextDataLogger() YDataLogger

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

```
function 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.

```
function 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() datalogger→setAutoStart()datalogger.set_autoStart() YDataLogger

Changes the default activation state of the data logger on power up.

```
function set_autoStart( newval)
```

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

Parameters :

newval either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the default activation state of the data logger on power up

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

datalogger→set_logicalName()
datalogger→setLogicalName()
datalogger.set_logicalName()

YDataLogger

Changes the logical name of the data logger.

```
function set_logicalName( newval)
```

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

Parameters :

newval a string corresponding to the logical name of the data logger.

Returns :

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

datalogger→set_recording() datalogger→setRecording() datalogger.set_recording() YDataLogger

Changes the activation state of the data logger to start/stop recording data.

```
function 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() datalogger→setTimeUTC()datalogger.set_timeUTC() YDataLogger

Changes the current UTC time reference used for recorded data.

```
function 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() datalogger→setUserData()datalogger.set_userData() YDataLogger

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

```
function 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

datalogger→wait_async()datalogger.wait_async() YDataLogger

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.9. Formatted data sequence

A run is a continuous interval of time during which a module was powered on. A data run provides easy access to all data collected during a given run, providing on-the-fly resampling at the desired reporting rate.

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

js	<script type='text/javascript' src='yocto_datalogger.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YDataLogger = yoctolib.YDataLogger;

php	require_once('yocto_datalogger.php');
cpp	#include "yocto_datalogger.h"
m	#import "yocto_datalogger.h"
pas	uses yocto_datalogger;
vb	yocto_datalogger.vb
cs	yocto_datalogger.cs
java	import com.yoctopuce.YoctoAPI.YDataLogger;
py	from yocto_datalogger import *

`YDataRun` methods

datarun→get_averageValue(measureName, pos)

Returns the average value of the measure observed at the specified time period.

datarun→get_duration()

Returns the duration (in seconds) of the data run.

datarun→get_maxValue(measureName, pos)

Returns the maximal value of the measure observed at the specified time period.

datarun→get_measureNames()

Returns the names of the measures recorded by the data logger.

datarun→get_minValue(measureName, pos)

Returns the minimal value of the measure observed at the specified time period.

datarun→get_startTimeUTC()

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

datarun→get_valueCount()

Returns the number of values accessible in this run, given the selected data samples interval.

datarun→get_valueInterval()

Returns the number of seconds covered by each value in this run.

datarun→set_valueInterval(valueInterval)

Changes the number of seconds covered by each value in this run.

datarun→get_averageValue() datarun→averageValue()datarun.get_averageValue() YDataRun

Returns the average value of the measure observed at the specified time period.

function **get_averageValue**(**measureName**, **pos**)

datarun→get_averageValue() datarun→averageValue()datarun.get_averageValue()

Returns the average value of the measure observed at the specified time period.

`js`	function **get_averageValue**(**measureName**, **pos**)
`nodejs`	function **get_averageValue**(**measureName**, **pos**)
`php`	function **get_averageValue**(**$measureName**, **$pos**)
`java`	double **get_averageValue**(String **measureName**, int **pos**)
`py`	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() datarun→duration()datarun.get_duration() YDataRun

Returns the duration (in seconds) of the data run.

function **get_duration**( )

datarun→get_duration() datarun→duration()datarun.get_duration()

Returns the duration (in seconds) of the data run.

`js`	function **get_duration**( )
`nodejs`	function **get_duration**( )
`php`	function **get_duration**( )
`java`	long **get_duration**( )
`py`	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() datarun→maxValue()datarun.get_maxValue() YDataRun

Returns the maximal value of the measure observed at the specified time period.

function **get_maxValue**(**measureName**, **pos**)

datarun→get_maxValue() datarun→maxValue()datarun.get_maxValue()

Returns the maximal value of the measure observed at the specified time period.

`js`	function **get_maxValue**(**measureName**, **pos**)
`nodejs`	function **get_maxValue**(**measureName**, **pos**)
`php`	function **get_maxValue**(**$measureName**, **$pos**)
`java`	double **get_maxValue**(String **measureName**, int **pos**)
`py`	def **get_maxValue**(**measureName**, **pos**)

Parameters :

measureName	the name of the desired measure (one of the names returned by `get_measureNames`)
pos	the position index, between 0 and the value returned by `get_valueCount`

Returns :

a floating point number (the maximal value)

On failure, throws an exception or returns Y_MAXVALUE_INVALID.

datarun→get_measureNames() datarun→measureNames() datarun.get_measureNames() YDataRun

Returns the names of the measures recorded by the data logger.

function **get_measureNames**( )

datarun→get_measureNames() datarun→measureNames()datarun.get_measureNames()

Returns the names of the measures recorded by the data logger.

js	function **get_measureNames**( )
nodejs	function **get_measureNames**( )
php	function **get_measureNames**( )
java	ArrayList<String> **get_measureNames**( )
py	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() datarun→minValue()datarun.get_minValue() YDataRun

Returns the minimal value of the measure observed at the specified time period.

function **get_minValue**(**measureName**, **pos**)

datarun→get_minValue() datarun→minValue()datarun.get_minValue()

Returns the minimal value of the measure observed at the specified time period.

`js`	function **get_minValue**(**measureName**, **pos**)
`nodejs`	function **get_minValue**(**measureName**, **pos**)
`php`	function **get_minValue**(**$measureName**, **$pos**)
`java`	double **get_minValue**(String **measureName**, int **pos**)
`py`	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() datarun→valueCount()datarun.get_valueCount() YDataRun

Returns the number of values accessible in this run, given the selected data samples interval.

function **get_valueCount**( )

datarun→get_valueCount() datarun→valueCount()datarun.get_valueCount()

Returns the number of values accessible in this run, given the selected data samples interval.

js	function **get_valueCount**( )
nodejs	function **get_valueCount**( )
php	function **get_valueCount**( )
java	int **get_valueCount**( )
py	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() datarun→valueInterval()datarun.get_valueInterval() — YDataRun

Returns the number of seconds covered by each value in this run.

function **get_valueInterval**( )

datarun→get_valueInterval() datarun→valueInterval()datarun.get_valueInterval()

Returns the number of seconds covered by each value in this run.

`js` function **get_valueInterval**( )
`nodejs` function **get_valueInterval**( )
`php` function **get_valueInterval**( )
`java` int **get_valueInterval**( )
`py` def **get_valueInterval**( )

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.

Returns :
an unsigned number corresponding to a number of seconds covered by each data sample in the Run.

datarun→set_valueInterval() datarun→setValueInterval() datarun.set_valueInterval()

YDataRun

Changes the number of seconds covered by each value in this run.

function **set_valueInterval**(**valueInterval**)

datarun→set_valueInterval() datarun→setValueInterval()datarun.set_valueInterval()

Changes the number of seconds covered by each value in this run.

`js`	function **set_valueInterval**(**valueInterval**)
`nodejs`	function **set_valueInterval**(**valueInterval**)
`php`	function **set_valueInterval**(**$valueInterval**)
`java`	void **set_valueInterval**(int **valueInterval**)
`py`	def **set_valueInterval**(**valueInterval**)

By default, the value interval is set to the coarsest data rate archived in the data logger flash for this run. The value interval can however be configured at will to a different rate when desired.

Parameters :

valueInterval an integer number of seconds.

Returns :

nothing

3.10. Recorded data sequence

YDataSet objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the YDataSet object is instanciated by the `get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the dataLogger.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

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

js	<script type='text/javascript' src='yocto_api.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YAPI = yoctolib.YAPI;
	var YModule = yoctolib.YModule;

php	require_once('yocto_api.php');
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

YDataSet methods
dataset→get_endTimeUTC() Returns the end time of the dataset, relative to the Jan 1, 1970.
dataset→get_functionId() Returns the hardware identifier of the function that performed the measure, without reference to the module.
dataset→get_hardwareId() Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.
dataset→get_measures() Returns all measured values currently available for this DataSet, as a list of YMeasure objects.
dataset→get_preview() Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.
dataset→get_progress() Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.
dataset→get_startTimeUTC() Returns the start time of the dataset, relative to the Jan 1, 1970.
dataset→get_summary() Returns an YMeasure object which summarizes the whole DataSet.
dataset→get_unit() Returns the measuring unit for the measured value.

dataset→loadMore()

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

dataset→loadMore_async(callback, context)

Loads the the next block of measures from the dataLogger asynchronously.

dataset→get_endTimeUTC() dataset→endTimeUTC()dataset.get_endTimeUTC() YDataSet

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

```
function 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() dataset→functionId()dataset.get_functionId() YDataSet

Returns the hardware identifier of the function that performed the measure, without reference to the module.

```
function get_functionId( )
```

For example `temperature1`.

Returns :

a string that identifies the function (ex: `temperature1`)

dataset→get_hardwareId() dataset→hardwareId()dataset.get_hardwareId() YDataSet

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

```
function 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() dataset→measures()dataset.get_measures() YDataSet

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

```
function 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() dataset→preview()dataset.get_preview() YDataSet

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

```
function 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() dataset→progress()dataset.get_progress() **YDataSet**

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

```
function get_progress( )
```

When the object is instanciated by `get_dataSet`, the progress is zero. Each time `loadMore()` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

Returns :
an integer in the range 0 to 100 (percentage of completion).

dataset→get_startTimeUTC() dataset→startTimeUTC()dataset.get_startTimeUTC() YDataSet

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

```
function 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() dataset→summary()dataset.get_summary() YDataSet

Returns an YMeasure object which summarizes the whole DataSet.

```
function 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()
dataset→unit()dataset.get_unit() YDataSet

Returns the measuring unit for the measured value.

```
function 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.

```
function 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.

dataset→loadMore_async()dataset.loadMore_async() YDataSet

Loads the the next block of measures from the dataLogger asynchronously.

```
function loadMore_async( callback, context)
```

Parameters :

callback	callback function that is invoked when the w The callback function receives three arguments: - the user-specific context object - the YDataSet object whose loadMore_async was invoked - the load result: either the progress indicator (0...100), or a negative error code in case of failure.
context	user-specific object that is passed as-is to the callback function

Returns :

nothing.

3.11. Unformatted data sequence

YDataStream objects represent bare recorded measure sequences, exactly as found within the data logger present on Yoctopuce sensors.

In most cases, it is not necessary to use YDataStream objects directly, as the YDataSet objects (returned by the `get_recordedData()` method from sensors and the `get_dataSets()` method from the data logger) provide a more convenient interface.

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

js	<script type='text/javascript' src='yocto_api.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YAPI = yoctolib.YAPI;
	var YModule = yoctolib.YModule;

php	require_once('yocto_api.php');
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

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

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

datastream→get_averageValue()
datastream→averageValue()
datastream.get_averageValue()

YDataStream

Returns the average of all measures observed within this stream.

```
function get_averageValue( )
```

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the average value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_columnCount() datastream→columnCount() datastream.get_columnCount() YDataStream

Returns the number of data columns present in this stream.

```
function get_columnCount( )
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :
an unsigned number corresponding to the number of columns.

On failure, throws an exception or returns zero.

datastream→get_columnNames() datastream→columnNames() datastream.get_columnNames() YDataStream

Returns the title (or meaning) of each data column present in this stream.

```
function 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() datastream→data()datastream.get_data() YDataStream

Returns a single measure from the data stream, specified by its row and column index.

```
function 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() datastream→dataRows()datastream.get_dataRows() YDataStream

Returns the whole data set contained in the stream, as a bidimensional table of numbers.

```
function get_dataRows( )
```

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

This method fetches the whole data stream from the device, if not yet done.

Returns :

a list containing as many elements as there are rows in the data stream. Each row itself is a list of floating-point numbers.

On failure, throws an exception or returns an empty array.

datastream→get_dataSamplesIntervalMs() datastream→dataSamplesIntervalMs() datastream.get_dataSamplesIntervalMs() YDataStream

Returns the number of milliseconds between two consecutive rows of this data stream.

```
function 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() datastream→duration()datastream.get_duration() YDataStream

Returns the approximate duration of this stream, in seconds.

```
function get_duration( )
```

Returns :
the number of seconds covered by this stream.

On failure, throws an exception or returns Y_DURATION_INVALID.

datastream→get_maxValue() datastream→maxValue()datastream.get_maxValue() YDataStream

Returns the largest measure observed within this stream.

```
function 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() datastream→minValue()datastream.get_minValue() YDataStream

Returns the smallest measure observed within this stream.

```
function 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() datastream→rowCount()datastream.get_rowCount() YDataStream

Returns the number of data rows present in this stream.

```
function 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() datastream→runIndex()datastream.get_runIndex() YDataStream

Returns the run index of the data stream.

```
function 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() datastream→startTime()datastream.get_startTime() YDataStream

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

```
function 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() datastream→startTimeUTC() datastream.get_startTimeUTC()

YDataStream

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

```
function 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).

3.12. Digital IO function interface

The Yoctopuce application programming interface allows you to switch the state of each bit of the I/O port. You can switch all bits at once, or one by one. The library can also automatically generate short pulses of a determined duration. Electrical behavior of each I/O can be modified (open drain and reverse polarity).

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

js	<script type='text/javascript' src='yocto_digitalio.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YDigitalIO = yoctolib.YDigitalIO;

php	require_once('yocto_digitalio.php');
cpp	#include "yocto_digitalio.h"
m	#import "yocto_digitalio.h"
pas	uses yocto_digitalio;
vb	yocto_digitalio.vb
cs	yocto_digitalio.cs
java	import com.yoctopuce.YoctoAPI.YDigitalIO;
py	from yocto_digitalio import *

Global functions

yFindDigitalIO(func)

Retrieves a digital IO port for a given identifier.

yFirstDigitalIO()

Starts the enumeration of digital IO ports currently accessible.

`YDigitalIO` methods

digitalio→delayedPulse(bitno, ms_delay, ms_duration)

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

digitalio→describe()

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

digitalio→get_advertisedValue()

Returns the current value of the digital IO port (no more than 6 characters).

digitalio→get_bitDirection(bitno)

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

digitalio→get_bitOpenDrain(bitno)

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

digitalio→get_bitPolarity(bitno)

Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).

digitalio→get_bitState(bitno)

Returns the state of a single bit of the I/O port.

digitalio→get_errorMessage()

Returns the error message of the latest error with the digital IO port.

digitalio→get_errorType()

Returns the numerical error code of the latest error with the digital IO port.

digitalio→get_friendlyName()

Returns a global identifier of the digital IO port in the format `MODULE_NAME.FUNCTION_NAME`.

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)

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()yFindDigitalIO() YDigitalIO

Retrieves a digital IO port for a given identifier.

```
function yFindDigitalIO( 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()yFirstDigitalIO() YDigitalIO

Starts the enumeration of digital IO ports currently accessible.

```
function yFirstDigitalIO( )
```

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.

```
function 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`.

```
function 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() digitalio→advertisedValue() digitalio.get_advertisedValue() YDigitalIO

Returns the current value of the digital IO port (no more than 6 characters).

function **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() digitalio→bitDirection()digitalio.get_bitDirection() YDigitalIO

Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).

```
function 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() digitalio→bitOpenDrain()digitalio.get_bitOpenDrain() YDigitalIO

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

```
function 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() digitalio→bitPolarity()digitalio.get_bitPolarity() YDigitalIO

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).

```
function 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() digitalio→bitState()digitalio.get_bitState() YDigitalIO

Returns the state of a single bit of the I/O port.

```
function get_bitState( bitno)
```

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

the bit state (0 or 1)

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

digitalio→get_errorMessage()
digitalio→errorMessage()
digitalio.get_errorMessage()

YDigitalIO

Returns the error message of the latest error with the digital IO port.

```
function get_errorMessage( )
```

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

Returns :

a string corresponding to the latest error message that occured while using the digital IO port object

digitalio→get_errorType()
digitalio→errorType()digitalio.get_errorType() YDigitalIO

Returns the numerical error code of the latest error with the digital IO port.

```
function 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 occured while using the digital IO port object

digitalio→get_friendlyName() digitalio→friendlyName()digitalio.get_friendlyName() YDigitalIO

Returns a global identifier of the digital IO port in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

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

Returns :

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

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

digitalio→get_functionDescriptor()
digitalio→functionDescriptor()
digitalio.get_functionDescriptor() YDigitalIO

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

```
function 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() digitalio→functionId()digitalio.get_functionId() YDigitalIO

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

```
function 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() digitalio→hardwareId()digitalio.get_hardwareId() YDigitalIO

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

```
function 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() digitalio→logicalName()digitalio.get_logicalName() YDigitalIO

Returns the logical name of the digital IO port.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the digital IO port. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

digitalio→get_module() digitalio→module()digitalio.get_module() YDigitalIO

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

```
function 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_module_async() digitalio→module_async() digitalio.get_module_async() YDigitalIO

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

digitalio→get_outputVoltage() digitalio→outputVoltage() digitalio.get_outputVoltage() YDigitalIO

Returns the voltage source used to drive output bits.

```
function 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() digitalio→portDirection()digitalio.get_portDirection() YDigitalIO

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

```
function 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.

```
function 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() digitalio→portPolarity()digitalio.get_portPolarity() YDigitalIO

Returns the polarity of all the bits of the port.

```
function 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() digitalio→portSize()digitalio.get_portSize() YDigitalIO

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

```
function 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() digitalio→portState()digitalio.get_portState() YDigitalIO

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

```
function 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() digitalio→userData()digitalio.get_userData() YDigitalIO

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

```
function 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.

```
function isOnline( )
```

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

Returns :

`true` if the digital IO port can be reached, and `false` otherwise

digitalio→isOnline_async()digitalio.isOnline_async() YDigitalIO

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

digitalio→load()digitalio.load() YDigitalIO

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

```
function 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 trafic 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→load_async()digitalio.load_async() YDigitalIO

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

digitalio→nextDigitalIO()digitalio.nextDigitalIO() YDigitalIO

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

```
function nextDigitalIO( )
```

Returns :

a pointer to a `YDigitalIO` object, corresponding to a digital IO port currently online, or a `null` pointer if there are no more digital IO ports to enumerate.

digitalio→pulse()digitalio.pulse() YDigitalIO

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

```
function 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.

```
function 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() digitalio→setBitDirection()digitalio.set_bitDirection() YDigitalIO

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

```
function set_bitDirection( bitno, bitdirection)
```

Parameters :

bitno	the bit number; lowest bit has index 0
bitdirection	direction to set, 0 makes the bit an input, 1 makes it an output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_bitOpenDrain()
digitalio→setBitOpenDrain()
digitalio.set_bitOpenDrain() YDigitalIO

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

function **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() digitalio→setBitPolarity()digitalio.set_bitPolarity() YDigitalIO

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

```
function set_bitPolarity( bitno, bitpolarity)
```

Parameters :

bitno	the bit number; lowest bit has index 0.
bitpolarity	polarity to set, 0 makes the I/O work in regular mode, 1 makes the I/O works in reverse mode. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_bitState() digitalio→setBitState()digitalio.set_bitState() YDigitalIO

Sets a single bit of the I/O port.

```
function set_bitState( bitno, bitstate)
```

Parameters :

bitno	the bit number; lowest bit has index 0
bitstate	the state of the bit (1 or 0)

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_logicalName()
digitalio→setLogicalName()
digitalio.set_logicalName() YDigitalIO

Changes the logical name of the digital IO port.

```
function set_logicalName( newval)
```

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

Parameters :

newval a string corresponding to the logical name of the digital IO port.

Returns :

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

digitalio→set_outputVoltage() digitalio→setOutputVoltage() digitalio.set_outputVoltage() — YDigitalIO

Changes the voltage source used to drive output bits.

```
function 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.

```
function 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.

```
function 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() digitalio→setPortPolarity()digitalio.set_portPolarity() YDigitalIO

Changes the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output.

function **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() digitalio→setPortState()digitalio.set_portState() YDigitalIO

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

```
function 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() digitalio→setUserData()digitalio.set_userData() YDigitalIO

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

```
function 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.

```
function 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.

digitalio→wait_async()digitalio.wait_async() YDigitalIO

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.13. Display function interface

Yoctopuce display interface has been designed to easily show information and images. The device provides built-in multi-layer rendering. Layers can be drawn offline, individually, and freely moved on the display. It can also replay recorded sequences (animations).

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

js	<script type='text/javascript' src='yocto_display.js'></script>

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 *

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→get_errorType()
Returns the numerical error code of the latest error with the display.

display→get_friendlyName()
Returns a global identifier of the display in the format `MODULE_NAME.FUNCTION_NAME`.

display→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

display→get_functionId()
Returns the hardware identifier of the display, without reference to the module.

display→get_hardwareId()
Returns the unique hardware identifier of the display in the form `SERIAL.FUNCTIONID`.

display→get_layerCount()
Returns the number of available layers to draw on.

display→get_layerHeight()
Returns the height of the layers to draw on, in pixels.

display→get_layerWidth()
Returns the width of the layers to draw on, in pixels.

display→get_logicalName()
Returns the logical name of the display.

display→get_module()
Gets the `YModule` object for the device on which the function is located.

display→get_module_async(callback, context)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

display→get_orientation()
Returns the currently selected display orientation.

display→get_startupSeq()
Returns the name of the sequence to play when the displayed is powered on.

display→get_userData()
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)

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()yFindDisplay() YDisplay

Retrieves a display for a given identifier.

```
function yFindDisplay( 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()yFirstDisplay() YDisplay

Starts the enumeration of displays currently accessible.

```
function yFirstDisplay( )
```

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.

```
function 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`.

```
function 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 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.

```
function fade( brightness, duration)
```

Parameters :

brightness	the new screen brightness
duration	duration of the brightness transition, in milliseconds.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

display→get_advertisedValue() display→advertisedValue() display.get_advertisedValue() YDisplay

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

function **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() display→brightness()display.get_brightness() YDisplay

Returns the luminosity of the module informative leds (from 0 to 100).

```
function 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() display→displayHeight()display.get_displayHeight() YDisplay

Returns the display height, in pixels.

```
function 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() display→displayLayer()display.get_displayLayer() YDisplay

Returns a YDisplayLayer object that can be used to draw on the specified layer.

```
function 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() display→displayType()display.get_displayType() YDisplay

Returns the display type: monochrome, gray levels or full color.

```
function 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() display→displayWidth()display.get_displayWidth() YDisplay

Returns the display width, in pixels.

```
function 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() display→enabled()display.get_enabled() YDisplay

Returns true if the screen is powered, false otherwise.

```
function 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() display→errorMessage()display.get_errorMessage() YDisplay

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

```
function get_errorMessage( )
```

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

Returns :
a string corresponding to the latest error message that occured while using the display object

display→get_errorType() display→errorType()display.get_errorType() YDisplay

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

```
function 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 occured while using the display object

display→get_friendlyName() display→friendlyName()display.get_friendlyName() YDisplay

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

```
function get_friendlyName( )
```

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

Returns :

a string that uniquely identifies the display using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

display→get_functionDescriptor() display→functionDescriptor() display.get_functionDescriptor() YDisplay

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

```
function get_functionDescriptor( )
```

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

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

display→get_functionId() display→functionId()display.get_functionId() YDisplay

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

```
function 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() display→hardwareId()display.get_hardwareId() YDisplay

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

```
function 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() display→layerCount()display.get_layerCount() YDisplay

Returns the number of available layers to draw on.

```
function get_layerCount( )
```

Returns :

an integer corresponding to the number of available layers to draw on

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

display→get_layerHeight() display→layerHeight()display.get_layerHeight() YDisplay

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

```
function 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() display→layerWidth()display.get_layerWidth() YDisplay

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

```
function 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() display→logicalName()display.get_logicalName() YDisplay

Returns the logical name of the display.

```
function 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() display→module()display.get_module() YDisplay

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

```
function 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_module_async() display→module_async() display.get_module_async() YDisplay

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

display→get_orientation() display→orientation()display.get_orientation() YDisplay

Returns the currently selected display orientation.

```
function 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() display→startupSeq()display.get_startupSeq() YDisplay

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

```
function 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() display→userData()display.get_userData() YDisplay

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

```
function 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.

```
function 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→isOnline_async()display.isOnline_async() YDisplay

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

display→load()display.load() YDisplay

Preloads the display cache with a specified validity duration.

```
function 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 trafic 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→load_async()display.load_async() YDisplay

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

display→newSequence()display.newSequence() YDisplay

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

```
function 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()`.

function **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.

```
function 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()`.

```
function 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.

```
function 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.

```
function resetAll( )
```

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

display→saveSequence()display.saveSequence() YDisplay

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

```
function saveSequence( sequenceName)
```

The sequence can be later replayed using `playSequence()`.

Parameters :

sequenceName the name of the newly created sequence

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

display→set_brightness() display→setBrightness()display.set_brightness() YDisplay

Changes the brightness of the display.

```
function 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() display→setEnabled()display.set_enabled() YDisplay

Changes the power state of the display.

```
function 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() display→setLogicalName()display.set_logicalName() YDisplay

Changes the logical name of the display.

```
function set_logicalName( newval)
```

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

Parameters :

newval a string corresponding to the logical name of the display.

Returns :

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

display→set_orientation() display→setOrientation()display.set_orientation() YDisplay

Changes the display orientation.

```
function 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() display→setStartupSeq()display.set_startupSeq() YDisplay

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

```
function 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() display→setUserData()display.set_userData() YDisplay

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

```
function 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.

```
function 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.

```
function 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 onae 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.

```
function 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.

display→wait_async()display.wait_async() YDisplay

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.14. DisplayLayer object interface

A DisplayLayer is an image layer containing objects to display (bitmaps, text, etc.). The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

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

js	`<script type='text/javascript' src='yocto_display.js'></script>`

nodejs	var yoctolib = require('yoctolib');
	var YDisplay = yoctolib.YDisplay;

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, bgcol)
Draws a bitmap at the specified position.

displaylayer→drawCircle(x, y, r)
Draws an empty circle at a specified position.

displaylayer→drawDisc(x, y, r)
Draws a filled disc at a given position.

displaylayer→drawImage(x, y, imagename)
Draws a GIF image at the specified position.

displaylayer→drawPixel(x, y)
Draws a single pixel at the specified position.

displaylayer→drawRect(x1, y1, x2, y2)
Draws an empty rectangle at a specified position.

displaylayer→drawText(x, y, anchor, text)
Draws a text string at the specified position.

displaylayer→get_display()
Gets parent YDisplay.

displaylayer→get_displayHeight()
Returns the display height, in pixels.

displaylayer→get_displayWidth()
Returns the display width, in pixels.

displaylayer→get_layerHeight()
Returns the height of the layers to draw on, in pixels.

displaylayer→get_layerWidth()
Returns the width of the layers to draw on, in pixels.

displaylayer→hide()
Hides the layer.

displaylayer→lineTo(**x**, **y**)
Draws a line from current drawing pointer position to the specified position.

displaylayer→moveTo(**x**, **y**)
Moves the drawing pointer of this layer to the specified position.

displaylayer→reset()
Reverts the layer to its initial state (fully transparent, default settings).

displaylayer→selectColorPen(**color**)
Selects the pen color for all subsequent drawing functions, including text drawing.

displaylayer→selectEraser()
Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

displaylayer→selectFont(**fontname**)
Selects a font to use for the next text drawing functions, by providing the name of the font file.

displaylayer→selectGrayPen(**graylevel**)
Selects the pen gray level for all subsequent drawing functions, including text drawing.

displaylayer→setAntialiasingMode(**mode**)
Enables or disables anti-aliasing for drawing oblique lines and circles.

displaylayer→setConsoleBackground(**bgcol**)
Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

displaylayer→setConsoleMargins(**x1**, **y1**, **x2**, **y2**)
Sets up display margins for the `consoleOut` function.

displaylayer→setConsoleWordWrap(**wordwrap**)
Sets up the wrapping behaviour used by the `consoleOut` function.

displaylayer→setLayerPosition(**x**, **y**, **scrollTime**)
Sets the position of the layer relative to the display upper left corner.

displaylayer→unhide()
Shows the layer.

displaylayer→clear()displaylayer.clear() YDisplayLayer

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

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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() displaylayer→display()displaylayer.get_display() YDisplayLayer

Gets parent YDisplay.

```
function get_display( )
```

Returns the parent YDisplay object of the current YDisplayLayer.

Returns :
an `YDisplay` object

displaylayer→get_displayHeight() displaylayer→displayHeight() displaylayer.get_displayHeight() YDisplayLayer

Returns the display height, in pixels.

```
function get_displayHeight( )
```

Returns :

an integer corresponding to the display height, in pixels On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.

displaylayer→get_displayWidth() displaylayer→displayWidth() displaylayer.get_displayWidth() YDisplayLayer

Returns the display width, in pixels.

```
function get_displayWidth( )
```

Returns :

an integer corresponding to the display width, in pixels On failure, throws an exception or returns Y_DISPLAYWIDTH_INVALID.

displaylayer→get_layerHeight() displaylayer→layerHeight() displaylayer.get_layerHeight() YDisplayLayer

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

```
function get_layerHeight( )
```

Returns :
an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERHEIGHT_INVALID.

displaylayer→get_layerWidth() displaylayer→layerWidth() displaylayer.get_layerWidth() YDisplayLayer

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

```
function 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.

```
function 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.

```
function 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.

```
function 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).

```
function 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.

```
function selectColorPen( color)
```

The pen color is provided as an RGB value. For grayscale or monochrome displays, the value is automatically converted to the proper range.

Parameters :

color the desired pen color, as a 24-bit RGB value

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→selectEraser()
displaylayer.selectEraser() YDisplayLayer

Selects an eraser instead of a pen for all subsequent drawing functions, except for text drawing and bitmap copy functions.

```
function 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.

```
function 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.

```
function selectGrayPen( graylevel)
```

The gray level is provided as a number between 0 (black) and 255 (white, or whichever the lighest 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function unhide( )
```

Shows the layer again after a hide command.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.15. External power supply control interface

Yoctopuce application programming interface allows you to control the power source to use for module functions that require high current. The module can also automatically disconnect the external power when a voltage drop is observed on the external power source (external battery running out of power).

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

`js`	<script type='text/javascript' src='yocto_dualpower.js'></script>

`nodejs`	var yoctolib = require('yoctolib');
	var YDualPower = yoctolib.YDualPower;

`php`	require_once('yocto_dualpower.php');
`cpp`	#include "yocto_dualpower.h"
`m`	#import "yocto_dualpower.h"
`pas`	uses yocto_dualpower;
`vb`	yocto_dualpower.vb
`cs`	yocto_dualpower.cs
`java`	import com.yoctopuce.YoctoAPI.YDualPower;
`py`	from yocto_dualpower import *

Global functions

yFindDualPower(**func**)

Retrieves a dual power control for a given identifier.

yFirstDualPower()

Starts the enumeration of dual power controls currently accessible.

`YDualPower` methods

dualpower→describe()

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

dualpower→get_advertisedValue()

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

dualpower→get_errorMessage()

Returns the error message of the latest error with the power control.

dualpower→get_errorType()

Returns the numerical error code of the latest error with the power control.

dualpower→get_extVoltage()

Returns the measured voltage on the external power source, in millivolts.

dualpower→get_friendlyName()

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

dualpower→get_functionDescriptor()

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

dualpower→get_functionId()

Returns the hardware identifier of the power control, without reference to the module.

dualpower→get_hardwareId()

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

dualpower→get_logicalName()

Returns the logical name of the power control.

dualpower→get_module()

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

dualpower→get_module_async(callback, context)

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

dualpower→get_powerControl()

Returns the selected power source for module functions that require lots of current.

dualpower→get_powerState()

Returns the current power source for module functions that require lots of current.

dualpower→get_userData()

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

dualpower→isOnline()

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

dualpower→isOnline_async(callback, context)

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

dualpower→load(msValidity)

Preloads the power control cache with a specified validity duration.

dualpower→load_async(msValidity, callback, context)

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

dualpower→nextDualPower()

Continues the enumeration of dual power controls started using `yFirstDualPower()`.

dualpower→registerValueCallback(callback)

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

dualpower→set_logicalName(newval)

Changes the logical name of the power control.

dualpower→set_powerControl(newval)

Changes the selected power source for module functions that require lots of current.

dualpower→set_userData(data)

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

dualpower→wait_async(callback, context)

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

YDualPower.FindDualPower() yFindDualPower()yFindDualPower() YDualPower

Retrieves a dual power control for a given identifier.

```
function yFindDualPower( func)
```

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the power control

Returns :

a `YDualPower` object allowing you to drive the power control.

YDualPower.FirstDualPower() yFirstDualPower()yFirstDualPower()

YDualPower

Starts the enumeration of dual power controls currently accessible.

```
function yFirstDualPower( )
```

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`.

```
function 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 power control (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

dualpower→get_advertisedValue() dualpower→advertisedValue() dualpower.get_advertisedValue() YDualPower

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

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the power control (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

dualpower→get_errorMessage() dualpower→errorMessage() dualpower.get_errorMessage() YDualPower

Returns the error message of the latest error with the power control.

```
function get_errorMessage( )
```

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

Returns :
a string corresponding to the latest error message that occured while using the power control object

dualpower→get_errorType() dualpower→errorType()dualpower.get_errorType() YDualPower

Returns the numerical error code of the latest error with the power control.

```
function 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 occured while using the power control object

dualpower→get_extVoltage() dualpower→extVoltage()dualpower.get_extVoltage() YDualPower

Returns the measured voltage on the external power source, in millivolts.

function **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`.

```
function get_friendlyName( )
```

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

Returns :

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

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

dualpower→get_functionDescriptor() dualpower→functionDescriptor() dualpower.get_functionDescriptor() YDualPower

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

```
function 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() dualpower→functionId()dualpower.get_functionId() YDualPower

Returns the hardware identifier of the power control, without reference to the module.

```
function 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() dualpower→hardwareId()dualpower.get_hardwareId() YDualPower

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

```
function 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() dualpower→logicalName() dualpower.get_logicalName() YDualPower

Returns the logical name of the power control.

```
function 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() dualpower→module()dualpower.get_module() YDualPower

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

```
function 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_module_async() dualpower→module_async() dualpower.get_module_async() YDualPower

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

dualpower→get_powerControl()
dualpower→powerControl()
dualpower.get_powerControl()

YDualPower

Returns the selected power source for module functions that require lots of current.

function **get_powerControl**( )

Returns :

a value among `Y_POWERCONTROL_AUTO`, `Y_POWERCONTROL_FROM_USB`, `Y_POWERCONTROL_FROM_EXT` and `Y_POWERCONTROL_OFF` corresponding to the selected power source for module functions that require lots of current

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

dualpower→get_powerState() dualpower→powerState() dualpower.get_powerState() YDualPower

Returns the current power source for module functions that require lots of current.

```
function 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() dualpower→userData()dualpower.get_userData() YDualPower

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

```
function 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.

```
function 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→isOnline_async() dualpower.isOnline_async() YDualPower

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

dualpower→load()dualpower.load() YDualPower

Preloads the power control cache with a specified validity duration.

```
function 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 trafic 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→load_async()dualpower.load_async() YDualPower

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

function **load_async**(**msValidity**, **callback**, **context**)

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

dualpower→nextDualPower() dualpower.nextDualPower() YDualPower

Continues the enumeration of dual power controls started using `yFirstDualPower()`.

```
function 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.

function **registerValueCallback**(**callback**)

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

Parameters :

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

dualpower→set_logicalName() dualpower→setLogicalName() dualpower.set_logicalName()

YDualPower

Changes the logical name of the power control.

```
function 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.

```
function 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() dualpower→setUserData()dualpower.set_userData() YDualPower

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

```
function 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

dualpower→wait_async()dualpower.wait_async() YDualPower

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.16. Files function interface

The filesystem interface makes it possible to store files on some devices, for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

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

js	<script type='text/javascript' src='yocto_files.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YFiles = yoctolib.YFiles;

php	require_once('yocto_files.php');
cpp	#include "yocto_files.h"
m	#import "yocto_files.h"
pas	uses yocto_files;
vb	yocto_files.vb
cs	yocto_files.cs
java	import com.yoctopuce.YoctoAPI.YFiles;
py	from yocto_files import *

Global functions

yFindFiles(func)

Retrieves a filesystem for a given identifier.

yFirstFiles()

Starts the enumeration of filesystems currently accessible.

`YFiles` methods

files→describe()

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

files→download(pathname)

Downloads the requested file and returns a binary buffer with its content.

files→download_async(pathname, callback, context)

Downloads the requested file and returns a binary buffer with its content.

files→format_fs()

Reinitializes the filesystem to its clean, unfragmented, empty state.

files→get_advertisedValue()

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

files→get_errorMessage()

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

files→get_errorType()

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

files→get_filesCount()

Returns the number of files currently loaded in the filesystem.

files→get_freeSpace()

Returns the free space for uploading new files to the filesystem, in bytes.

files→get_friendlyName()

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

files→get_functionDescriptor()

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

files→get_functionId()

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()yFindFiles() YFiles

Retrieves a filesystem for a given identifier.

```
function yFindFiles( 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() yFirstFiles()yFirstFiles() YFiles

Starts the enumeration of filesystems currently accessible.

```
function yFirstFiles( )
```

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`.

```
function 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 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.

```
function 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→download_async()files.download_async() YFiles

Downloads the requested file and returns a binary buffer with its content.

```
function download_async( pathname, callback, context)
```

This is the asynchronous version that uses a callback to pass the result when the donwload is completed.

Parameters :

pathname	path and name of the new file to load
callback	callback function that is invoked when the w The callback function receives three arguments: - the user-specific context object - the YFiles object whose download_async was invoked - a binary buffer with the file content
context	user-specific object that is passed as-is to the callback function

Returns :

nothing.

files→format_fs()files.format_fs() YFiles

Reinitializes the filesystem to its clean, unfragmented, empty state.

```
function 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() files→advertisedValue()files.get_advertisedValue() YFiles

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

```
function 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() files→errorMessage()files.get_errorMessage() YFiles

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

```
function get_errorMessage( )
```

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

Returns :
a string corresponding to the latest error message that occured while using the filesystem object

files→get_errorType() files→errorType()files.get_errorType() YFiles

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

```
function 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 occured while using the filesystem object

files→get_filesCount() files→filesCount()files.get_filesCount() YFiles

Returns the number of files currently loaded in the filesystem.

```
function 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() files→freeSpace()files.get_freeSpace() YFiles

Returns the free space for uploading new files to the filesystem, in bytes.

```
function 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() files→friendlyName()files.get_friendlyName() YFiles

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

```
function get_friendlyName( )
```

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

Returns :

a string that uniquely identifies the filesystem using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

files→get_functionDescriptor() files→functionDescriptor() files.get_functionDescriptor() YFiles

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

```
function 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() files→functionId()files.get_functionId() YFiles

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

```
function 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() files→hardwareId()files.get_hardwareId() YFiles

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

```
function 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() YFiles

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

```
function 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() files→logicalName()files.get_logicalName() YFiles

Returns the logical name of the filesystem.

```
function 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() files→module()files.get_module() YFiles

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

```
function 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_module_async() files→module_async()files.get_module_async() YFiles

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

files→get_userData() files→userData()files.get_userData() YFiles

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

```
function 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.

```
function 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→isOnline_async()files.isOnline_async() YFiles

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

files→load()files.load() YFiles

Preloads the filesystem cache with a specified validity duration.

```
function 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 trafic 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→load_async()files.load_async() YFiles

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

files→nextFiles()files.nextFiles() YFiles

Continues the enumeration of filesystems started using `yFirstFiles()`.

```
function 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.

```
function 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.

```
function 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() files→setLogicalName()files.set_logicalName() YFiles

Changes the logical name of the filesystem.

```
function 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() files→setUserData()files.set_userData() YFiles

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

```
function 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.

```
function 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.

files→wait_async()files.wait_async() YFiles

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.17. GenericSensor function interface

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

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

js	<script type='text/javascript' src='yocto_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_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()`.

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_signalRange(newval)
Changes the electric signal range used by the sensor.

genericsensor→set_unit(newval)
Changes the measuring unit for the measured value.

genericsensor→set_userData(data)
Stores a user context provided as argument in the userData attribute of the function.

genericsensor→set_valueRange(newval)
Changes the physical value range measured by the sensor.

genericsensor→wait_async(callback, context)
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YGenericSensor.FindGenericSensor() yFindGenericSensor()yFindGenericSensor() YGenericSensor

Retrieves a generic sensor for a given identifier.

```
function yFindGenericSensor( 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()yFirstGenericSensor() — YGenericSensor

Starts the enumeration of generic sensors currently accessible.

```
function yFirstGenericSensor( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

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

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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`.

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

genericsensor→get_advertisedValue() genericsensor→advertisedValue() genericsensor.get_advertisedValue() YGenericSensor

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

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the generic sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

genericsensor→get_currentRawValue() genericsensor→currentRawValue() genericsensor.get_currentRawValue() YGenericSensor

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

```
function get_currentRawValue( )
```

Returns :

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

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

genericsensor→get_currentValue() genericsensor→currentValue() genericsensor.get_currentValue() YGenericSensor

Returns the current measured value.

```
function 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.

```
function get_errorMessage( )
```

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

Returns :
a string corresponding to the latest error message that occured while using 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.

```
function 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 occured 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`.

```
function get_friendlyName( )
```

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

Returns :

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

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

genericsensor→get_functionDescriptor() genericsensor→functionDescriptor() genericsensor.get_functionDescriptor() YGenericSensor

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

```
function 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.

```
function 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`.

```
function 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.

```
function get_highestValue( )
```

Returns :

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

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

genericsensor→get_logFrequency() genericsensor→logFrequency() genericsensor.get_logFrequency() YGenericSensor

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

```
function 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.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the generic sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

genericsensor→get_lowestValue() genericsensor→lowestValue() genericsensor.get_lowestValue() YGenericSensor

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

```
function 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.

```
function 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_module_async()
genericsensor→module_async()
genericsensor.get_module_async()

YGenericSensor

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

genericsensor→get_recordedData() genericsensor→recordedData() genericsensor.get_recordedData()

YGenericSensor

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

```
function 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() genericsensor→reportFrequency() genericsensor.get_reportFrequency() YGenericSensor

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

```
function 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.

```
function get_resolution( )
```

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

Returns :

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

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

genericsensor→get_signalRange() genericsensor→signalRange() genericsensor.get_signalRange() YGenericSensor

Returns the electric signal range used by the sensor.

```
function 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.

```
function get_signalUnit( )
```

Returns :

a string corresponding to the measuring unit of the electrical signal used by the sensor

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

genericsensor→get_signalValue()
genericsensor→signalValue()
genericsensor.get_signalValue() YGenericSensor

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

```
function 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() genericsensor→unit()genericsensor.get_unit() YGenericSensor

Returns the measuring unit for the measure.

```
function get_unit( )
```

Returns :
a string corresponding to the measuring unit for the measure

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

genericsensor→get_userData() genericsensor→userData() genericsensor.get_userData() YGenericSensor

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

```
function 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.

```
function 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.

```
function 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→isOnline_async() genericsensor.isOnline_async() YGenericSensor

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

genericsensor→load()genericsensor.load() YGenericSensor

Preloads the generic sensor cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async() genericsensor.load_async() — YGenericSensor

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

genericsensor→nextGenericSensor() genericsensor.nextGenericSensor() YGenericSensor

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

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function set_logFrequency( newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

genericsensor→set_logicalName() genericsensor→setLogicalName() genericsensor.set_logicalName() — YGenericSensor

Changes the logical name of the generic sensor.

```
function 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.

```
function 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.

```
function 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.

```
function set_resolution( newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

genericsensor→set_signalRange() genericsensor→setSignalRange() genericsensor.set_signalRange() YGenericSensor

Changes the electric signal range used by the sensor.

```
function 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() genericsensor→setUnit()genericsensor.set_unit() YGenericSensor

Changes the measuring unit for the measured value.

```
function 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.

```
function 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.

```
function set_valueRange( newval)
```

The range change may have a side effect on the display resolution, as it may be adapted automatically.

Parameters :

newval a string corresponding to the physical value range measured by the sensor

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

genericsensor→wait_async() **genericsensor.wait_async()** **YGenericSensor**

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.18. Gyroscope function interface

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

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

js	<script type='text/javascript' src='yocto_gyro.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YGyro = yoctolib.YGyro;

php	require_once('yocto_gyro.php');
cpp	#include "yocto_gyro.h"
m	#import "yocto_gyro.h"
pas	uses yocto_gyro;
vb	yocto_gyro.vb
cs	yocto_gyro.cs
java	import com.yoctopuce.YoctoAPI.YGyro;
py	from yocto_gyro import *

Global functions

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.

gyro→get_currentValue()

Returns the current value of the angular velocity.

gyro→get_errorMessage()

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

gyro→get_errorType()

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

gyro→get_friendlyName()

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

gyro→get_functionDescriptor()

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

gyro→get_functionId()

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

gyro→get_hardwareId()

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

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 `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_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()yFindGyro() YGyro

Retrieves a gyroscope for a given identifier.

```
function yFindGyro( 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()yFirstGyro()

YGyro

Starts the enumeration of gyroscopes currently accessible.

```
function yFirstGyro( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

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

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

gyro→describe()gyro.describe() YGyro

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

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

gyro→get_advertisedValue() gyro→advertisedValue()gyro.get_advertisedValue() YGyro

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

```
function 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() gyro→currentRawValue()gyro.get_currentRawValue() YGyro

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

```
function get_currentRawValue( )
```

Returns :

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

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

gyro→get_currentValue() gyro→currentValue()gyro.get_currentValue() YGyro

Returns the current value of the angular velocity.

```
function get_currentValue( )
```

Returns :
a floating point number corresponding to the current value of the angular velocity

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

gyro→get_errorMessage() gyro→errorMessage()gyro.get_errorMessage() YGyro

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

```
function get_errorMessage( )
```

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

Returns :
a string corresponding to the latest error message that occured while using the gyroscope object

gyro→get_errorType() gyro→errorType()gyro.get_errorType() YGyro

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

```
function 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 occured while using the gyroscope object

gyro→get_friendlyName() gyro→friendlyName()gyro.get_friendlyName() YGyro

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

```
function get_friendlyName( )
```

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

Returns :

a string that uniquely identifies the gyroscope using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

gyro→get_functionDescriptor() gyro→functionDescriptor() gyro.get_functionDescriptor() YGyro

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

```
function 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() gyro→functionId()gyro.get_functionId() YGyro

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

```
function 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() gyro→hardwareId()gyro.get_hardwareId() YGyro

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

```
function 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() gyro→heading()gyro.get_heading() YGyro

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

```
function 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() gyro→highestValue()gyro.get_highestValue() YGyro

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

```
function 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() gyro→logFrequency()gyro.get_logFrequency() YGyro

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

```
function 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() gyro→logicalName()gyro.get_logicalName() YGyro

Returns the logical name of the gyroscope.

```
function 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() gyro→lowestValue()gyro.get_lowestValue() YGyro

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

```
function 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() gyro→module()gyro.get_module() YGyro

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

```
function 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_module_async() gyro→module_async()gyro.get_module_async() YGyro

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

gyro→get_pitch() gyro→pitch()gyro.get_pitch() YGyro

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

```
function 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() gyro→quaternionW()gyro.get_quaternionW() YGyro

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.

```
function get_quaternionW( )
```

Returns :

a floating-point number corresponding to the `w` component of the quaternion.

gyro→get_quaternionX() gyro→quaternionX()gyro.get_quaternionX() YGyro

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.

```
function 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() gyro→quaternionY()gyro.get_quaternionY() YGyro

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.

```
function 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() gyro→quaternionZ()gyro.get_quaternionZ() YGyro

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.

```
function 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() gyro→recordedData()gyro.get_recordedData() YGyro

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

```
function get_recordedData( startTime, endTime)
```

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

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

Parameters :

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

Returns :

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

gyro→get_reportFrequency() gyro→reportFrequency()gyro.get_reportFrequency() YGyro

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

```
function 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() gyro→resolution()gyro.get_resolution() YGyro

Returns the resolution of the measured values.

```
function 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() gyro→roll()gyro.get_roll() YGyro

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

```
function 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() gyro→unit()gyro.get_unit() **YGyro**

Returns the measuring unit for the angular velocity.

```
function 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() gyro→userData()gyro.get_userData() YGyro

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

```
function 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() gyro→xValue()gyro.get_xValue() YGyro

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

```
function 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() gyro→yValue()gyro.get_yValue() YGyro

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

```
function 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() gyro→zValue()gyro.get_zValue() YGyro

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

```
function 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.

```
function 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→isOnline_async()gyro.isOnline_async() YGyro

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

gyro→load()gyro.load() YGyro

Preloads the gyroscope cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async()gyro.load_async() YGyro

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

function **load_async**(**msValidity**, **callback**, **context**)

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

gyro→nextGyro()gyro.nextGyro() YGyro

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

```
function 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.

```
function 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.

```
function 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.

```
function 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.

```
function 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() gyro→setHighestValue()gyro.set_highestValue() YGyro

Changes the recorded maximal value observed.

```
function 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() gyro→setLogFrequency()gyro.set_logFrequency() YGyro

Changes the datalogger recording frequency for this function.

```
function 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() gyro→setLogicalName()gyro.set_logicalName() YGyro

Changes the logical name of the gyroscope.

```
function 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() gyro→setLowestValue()gyro.set_lowestValue() YGyro

Changes the recorded minimal value observed.

```
function 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.

```
function 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() gyro→setResolution()gyro.set_resolution() YGyro

Changes the resolution of the measured physical values.

```
function 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() gyro→setUserData()gyro.set_userData() YGyro

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

```
function 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

gyro→wait_async()gyro.wait_async() **YGyro**

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.19. Yocto-hub port interface

YHubPort objects provide control over the power supply for every YoctoHub port and provide information about the device connected to it. The logical name of a YHubPort is always automatically set to the unique serial number of the Yoctopuce device connected to it.

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

js	<script type='text/javascript' src='yocto_hubport.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YHubPort = yoctolib.YHubPort;

php	require_once('yocto_hubport.php');
cpp	#include "yocto_hubport.h"
m	#import "yocto_hubport.h"
pas	uses yocto_hubport;
vb	yocto_hubport.vb
cs	yocto_hubport.cs
java	import com.yoctopuce.YoctoAPI.YHubPort;
py	from yocto_hubport import *

Global functions
yFindHubPort(func) Retrieves a Yocto-hub port for a given identifier.
yFirstHubPort() Starts the enumeration of Yocto-hub ports currently accessible.
`YHubPort` methods
hubport→describe() Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
hubport→get_advertisedValue() Returns the current value of the Yocto-hub port (no more than 6 characters).
hubport→get_baudRate() Returns the current baud rate used by this Yocto-hub port, in kbps.
hubport→get_enabled() Returns true if the Yocto-hub port is powered, false otherwise.
hubport→get_errorMessage() Returns the error message of the latest error with the Yocto-hub port.
hubport→get_errorType() Returns the numerical error code of the latest error with the Yocto-hub port.
hubport→get_friendlyName() Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME.FUNCTION_NAME`.
hubport→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
hubport→get_functionId() Returns the hardware identifier of the Yocto-hub port, without reference to the module.
hubport→get_hardwareId() Returns the unique hardware identifier of the Yocto-hub port in the form `SERIAL.FUNCTIONID`.
hubport→get_logicalName() Returns the logical name of the Yocto-hub port.

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() yFindHubPort()yFindHubPort() YHubPort

Retrieves a Yocto-hub port for a given identifier.

```
function yFindHubPort( 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()yFirstHubPort() **YHubPort**

Starts the enumeration of Yocto-hub ports currently accessible.

```
function yFirstHubPort( )
```

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`.

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

hubport→get_advertisedValue() hubport→advertisedValue() hubport.get_advertisedValue() YHubPort

Returns the current value of the Yocto-hub port (no more than 6 characters).

```
function 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() hubport→baudRate()hubport.get_baudRate() YHubPort

Returns the current baud rate used by this Yocto-hub port, in kbps.

```
function 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() hubport→enabled()hubport.get_enabled() YHubPort

Returns true if the Yocto-hub port is powered, false otherwise.

```
function 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() hubport→errorMessage()hubport.get_errorMessage() YHubPort

Returns the error message of the latest error with the Yocto-hub port.

```
function get_errorMessage( )
```

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

Returns :
a string corresponding to the latest error message that occured while using the Yocto-hub port object

hubport→get_errorType() hubport→errorType()hubport.get_errorType() YHubPort

Returns the numerical error code of the latest error with the Yocto-hub port.

```
function 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 occured while using the Yocto-hub port object

hubport→get_friendlyName() hubport→friendlyName()hubport.get_friendlyName() YHubPort

Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

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

Returns :

a string that uniquely identifies the Yocto-hub port using logical names (ex: `MyCustomName.relay1`)

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

hubport→get_functionDescriptor() hubport→functionDescriptor() hubport.get_functionDescriptor() — YHubPort

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

```
function 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() hubport→functionId()hubport.get_functionId() YHubPort

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

```
function 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() hubport→hardwareId()hubport.get_hardwareId() YHubPort

Returns the unique hardware identifier of the Yocto-hub port in the form `SERIAL.FUNCTIONID`.

```
function 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() hubport→logicalName()hubport.get_logicalName() YHubPort

Returns the logical name of the Yocto-hub port.

```
function 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() hubport→module()hubport.get_module() YHubPort

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

```
function 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_module_async() hubport→module_async() hubport.get_module_async() YHubPort

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

hubport→get_portState() hubport→portState()hubport.get_portState() YHubPort

Returns the current state of the Yocto-hub port.

```
function 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() hubport→userData()hubport.get_userData() YHubPort

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

function **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.

```
function 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→isOnline_async()hubport.isOnline_async() YHubPort

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

hubport→load()hubport.load() YHubPort

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

```
function 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 trafic 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→load_async()hubport.load_async() YHubPort

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

hubport→nextHubPort()hubport.nextHubPort() YHubPort

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

```
function 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.

function **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() hubport→setEnabled()hubport.set_enabled() YHubPort

Changes the activation of the Yocto-hub port.

```
function set_enabled( newval)
```

If the port is enabled, the connected module is powered. Otherwise, port power is shut down.

Parameters :

newval either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the activation of the Yocto-hub port

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

hubport→set_logicalName() hubport→setLogicalName() hubport.set_logicalName() — YHubPort

Changes the logical name of the Yocto-hub port.

```
function 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() hubport→setUserData()hubport.set_userData() YHubPort

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

```
function 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

hubport→wait_async()hubport.wait_async() YHubPort

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.20. Humidity function interface

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

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

js	<script type='text/javascript' src='yocto_humidity.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YHumidity = yoctolib.YHumidity;

php	require_once('yocto_humidity.php');
cpp	#include "yocto_humidity.h"
m	#import "yocto_humidity.h"
pas	uses yocto_humidity;
vb	yocto_humidity.vb
cs	yocto_humidity.cs
java	import com.yoctopuce.YoctoAPI.YHumidity;
py	from yocto_humidity import *

Global functions

yFindHumidity(func)

Retrieves a humidity sensor for a given identifier.

yFirstHumidity()

Starts the enumeration of humidity sensors currently accessible.

`YHumidity` methods

humidity→calibrateFromPoints(rawValues, refValues)

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

humidity→describe()

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

humidity→get_advertisedValue()

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

humidity→get_currentRawValue()

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

humidity→get_currentValue()

Returns the current measure for the humidity.

humidity→get_errorMessage()

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

humidity→get_errorType()

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

humidity→get_friendlyName()

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

humidity→get_functionDescriptor()

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

humidity→get_functionId()

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

humidity→get_hardwareId()

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

humidity→get_highestValue()
: Returns the maximal value observed for the humidity.

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.

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 for the humidity.

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 for the humidity.

humidity→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

humidity→set_resolution(newval)

Changes the resolution of the measured physical values.

humidity→set_userData(data)

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

humidity→wait_async(callback, context)

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

YHumidity.FindHumidity() yFindHumidity()yFindHumidity() — YHumidity

Retrieves a humidity sensor for a given identifier.

```
function yFindHumidity( 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()yFirstHumidity() YHumidity

Starts the enumeration of humidity sensors currently accessible.

```
function yFirstHumidity( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

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

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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`.

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

humidity→get_advertisedValue() humidity→advertisedValue() humidity.get_advertisedValue() YHumidity

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

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the humidity sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

humidity→get_currentRawValue() humidity→currentRawValue() humidity.get_currentRawValue() YHumidity

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

```
function get_currentRawValue( )
```

Returns :

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

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

humidity→get_currentValue() humidity→currentValue()humidity.get_currentValue() YHumidity

Returns the current measure for the humidity.

```
function get_currentValue( )
```

Returns :
a floating point number corresponding to the current measure for the humidity

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

humidity→get_errorMessage()
humidity→errorMessage()
humidity.get_errorMessage()

YHumidity

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

```
function get_errorMessage( )
```

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

Returns :

a string corresponding to the latest error message that occured while using the humidity sensor object

humidity→get_errorType() humidity→errorType()humidity.get_errorType() YHumidity

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

```
function 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 occured while using the humidity sensor object

humidity→get_friendlyName() humidity→friendlyName() humidity.get_friendlyName()

YHumidity

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

```
function get_friendlyName( )
```

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

Returns :

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

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

humidity→get_functionDescriptor() humidity→functionDescriptor() humidity.get_functionDescriptor() YHumidity

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

```
function 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() humidity→functionId()humidity.get_functionId() YHumidity

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

```
function 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() humidity→hardwareId()humidity.get_hardwareId() YHumidity

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

```
function 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() humidity→highestValue() humidity.get_highestValue() YHumidity

Returns the maximal value observed for the humidity.

```
function get_highestValue( )
```

Returns :

a floating point number corresponding to the maximal value observed for the humidity

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

humidity→get_logFrequency()
humidity→logFrequency()
humidity.get_logFrequency()

YHumidity

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

```
function 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() humidity→logicalName()humidity.get_logicalName() YHumidity

Returns the logical name of the humidity sensor.

```
function 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() humidity→lowestValue()humidity.get_lowestValue() YHumidity

Returns the minimal value observed for the humidity.

```
function get_lowestValue( )
```

Returns :

a floating point number corresponding to the minimal value observed for the humidity

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

humidity→get_module() humidity→module()humidity.get_module() YHumidity

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

```
function 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_module_async() humidity→module_async() humidity.get_module_async() YHumidity

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

humidity→get_recordedData()
humidity→recordedData()
humidity.get_recordedData()

YHumidity

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

```
function 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.

humidity→get_reportFrequency() humidity→reportFrequency() humidity.get_reportFrequency()   YHumidity

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

```
function 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() humidity→resolution()humidity.get_resolution() YHumidity

Returns the resolution of the measured values.

```
function 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() humidity→unit()humidity.get_unit() YHumidity

Returns the measuring unit for the humidity.

```
function 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() humidity→userData()humidity.get_userData() YHumidity

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

```
function 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.

```
function 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→isOnline_async() humidity.isOnline_async() YHumidity

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

humidity→load()humidity.load() YHumidity

Preloads the humidity sensor cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async()humidity.load_async() YHumidity

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

humidity→nextHumidity()humidity.nextHumidity() YHumidity

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

```
function 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.

```
function 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.

```
function registerValueCallback( callback)
```

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

Parameters :

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

humidity→set_highestValue() humidity→setHighestValue() humidity.set_highestValue() YHumidity

Changes the recorded maximal value observed for the humidity.

```
function set_highestValue( newval)
```

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

humidity→set_logFrequency()
humidity→setLogFrequency()
humidity.set_logFrequency() YHumidity

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

humidity→set_logicalName() humidity→setLogicalName() humidity.set_logicalName() YHumidity

Changes the logical name of the humidity sensor.

```
function set_logicalName( newval)
```

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

Parameters :

newval a string corresponding to the logical name of the humidity sensor.

Returns :

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

humidity→set_lowestValue()
humidity→setLowestValue()
humidity.set_lowestValue()

YHumidity

Changes the recorded minimal value observed for the humidity.

```
function set_lowestValue( newval)
```

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

humidity→set_reportFrequency() humidity→setReportFrequency() humidity.set_reportFrequency() YHumidity

Changes the timed value notification frequency for this function.

```
function 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() humidity→setResolution()humidity.set_resolution() YHumidity

Changes the resolution of the measured physical values.

```
function 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() humidity→setUserData()humidity.set_userData() YHumidity

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

```
function 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

humidity→wait_async()humidity.wait_async() YHumidity

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.21. Led function interface

Yoctopuce application programming interface allows you not only to drive the intensity of the led, but also to have it blink at various preset frequencies.

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

js	<script type='text/javascript' src='yocto_led.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YLed = yoctolib.YLed;

php	require_once('yocto_led.php');
cpp	#include "yocto_led.h"
m	#import "yocto_led.h"
pas	uses yocto_led;
vb	yocto_led.vb
cs	yocto_led.cs
java	import com.yoctopuce.YoctoAPI.YLed;
py	from yocto_led import *

Global functions

yFindLed(**func**)
    Retrieves a led for a given identifier.

yFirstLed()
    Starts the enumeration of leds currently accessible.

YLed methods

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

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

led→get_blinking()
    Returns the current led signaling mode.

led→get_errorMessage()
    Returns the error message of the latest error with the led.

led→get_errorType()
    Returns the numerical error code of the latest error with the led.

led→get_friendlyName()
    Returns a global identifier of the led in the format `MODULE_NAME.FUNCTION_NAME`.

led→get_functionDescriptor()
    Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

led→get_functionId()
    Returns the hardware identifier of the led, without reference to the module.

led→get_hardwareId()
    Returns the unique hardware identifier of the led in the form `SERIAL.FUNCTIONID`.

led→get_logicalName()
    Returns the logical name of the led.

led→get_luminosity()
    Returns the current led intensity (in per cent).

led→get_module()

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

led→get_module_async(callback, context)

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

led→get_power()

Returns the current led state.

led→get_userData()

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

led→isOnline()

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

led→isOnline_async(callback, context)

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

led→load(msValidity)

Preloads the led cache with a specified validity duration.

led→load_async(msValidity, callback, context)

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

led→nextLed()

Continues the enumeration of leds started using `yFirstLed()`.

led→registerValueCallback(callback)

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

led→set_blinking(newval)

Changes the current led signaling mode.

led→set_logicalName(newval)

Changes the logical name of the led.

led→set_luminosity(newval)

Changes the current led intensity (in per cent).

led→set_power(newval)

Changes the state of the led.

led→set_userData(data)

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

led→wait_async(callback, context)

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

YLed.FindLed() yFindLed()yFindLed() YLed

Retrieves a led for a given identifier.

```
function yFindLed( 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.FirstL ed() yFirstLed()yFirstLed() YLed

Starts the enumeration of leds currently accessible.

```
function yFirstLed( )
```

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`.

```
function 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() led→advertisedValue()led.get_advertisedValue() YLed

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

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the led (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

led→get_blinking() led→blinking()led.get_blinking() YLed

Returns the current led signaling mode.

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

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

```
function get_errorMessage( )
```

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

Returns :
a string corresponding to the latest error message that occured while using the led object

led→get_errorType() led→errorType()led.get_errorType() YLed

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

```
function 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 occured while using the led object

led→get_friendlyName() led→friendlyName()led.get_friendlyName() YLed

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

```
function get_friendlyName( )
```

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

Returns :

a string that uniquely identifies the led using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

led→get_functionDescriptor() led→functionDescriptor() led.get_functionDescriptor() YLed

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

```
function 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() led→functionId()led.get_functionId() **YLed**

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

```
function 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() led→hardwareId()led.get_hardwareId() YLed

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

```
function 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() led→logicalName()led.get_logicalName() YLed

Returns the logical name of the led.

```
function 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() led→luminosity()led.get_luminosity() YLed

Returns the current led intensity (in per cent).

```
function 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() led→module()led.get_module() YLed

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

```
function 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_module_async() led→module_async()led.get_module_async() YLed

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

led→get_power() led→power()led.get_power() YLed

Returns the current led state.

```
function 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() led→userData()led.get_userData() YLed

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

function **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.

```
function 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→isOnline_async()led.isOnline_async() YLed

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

led→load()led.load() YLed

Preloads the led cache with a specified validity duration.

```
function 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 trafic 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→load_async()led.load_async() YLed

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

led→nextLed()led.nextLed() YLed

Continues the enumeration of leds started using `yFirstLed()`.

```
function 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.

```
function 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() led→setBlinking()led.set_blinking() YLed

Changes the current led signaling mode.

```
function 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() led→setLogicalName()led.set_logicalName() YLed

Changes the logical name of the led.

```
function 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() led→setLuminosity()led.set_luminosity() YLed

Changes the current led intensity (in per cent).

```
function 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() led→setPower()led.set_power() YLed

Changes the state of the led.

```
function set_power( newval)
```

Parameters :

newval either `Y_POWER_OFF` or `Y_POWER_ON`, according to the state of the led

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

led→set_userData() led→setUserData()led.set_userData() YLed

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

```
function 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

led→wait_async()led.wait_async() YLed

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.22. LightSensor function interface

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

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

js	<script type='text/javascript' src='yocto_lightsensor.js'></script>

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 unrounded and uncalibrated raw value returned by the sensor.

lightsensor→get_currentValue()
Returns the current measure for the ambiant light.

lightsensor→get_errorMessage()
Returns the error message of the latest error with the light sensor.

lightsensor→get_errorType()
Returns the numerical error code of the latest error with the light sensor.

lightsensor→get_friendlyName()
Returns a global identifier of the light sensor in the format `MODULE_NAME.FUNCTION_NAME`.

lightsensor→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

lightsensor→get_functionId()

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

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 ambiant light.

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 ambiant light.

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 ambiant 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 for the ambiant light.

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 for the ambiant light.

lightsensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

lightsensor→set_resolution(newval)

Changes the resolution of the measured physical values.

lightsensor→set_userData(data)

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

lightsensor→wait_async(callback, context)

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

YLightSensor.FindLightSensor() yFindLightSensor()yFindLightSensor() — YLightSensor

Retrieves a light sensor for a given identifier.

```
function yFindLightSensor( 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()yFirstLightSensor()

YLightSensor

Starts the enumeration of light sensors currently accessible.

```
function yFirstLightSensor( )
```

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).

```
function 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.

```
function calibrateFromPoints( rawValues, refValues)
```

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

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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`.

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

lightsensor→get_advertisedValue()
lightsensor→advertisedValue()
lightsensor.get_advertisedValue()

YLightSensor

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

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the light sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

lightsensor→get_currentRawValue() lightsensor→currentRawValue() lightsensor.get_currentRawValue() YLightSensor

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

```
function get_currentRawValue( )
```

Returns :

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

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

lightsensor→get_currentValue()
lightsensor→currentValue()
lightsensor.get_currentValue()

YLightSensor

Returns the current measure for the ambiant light.

```
function get_currentValue( )
```

Returns :

a floating point number corresponding to the current measure for the ambiant light

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

lightsensor→get_errorMessage()
lightsensor→errorMessage()
lightsensor.get_errorMessage() YLightSensor

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

```
function get_errorMessage( )
```

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

Returns :

a string corresponding to the latest error message that occured while using the light sensor object

lightsensor→get_errorType() lightsensor→errorType()lightsensor.get_errorType() YLightSensor

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

```
function 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 occured while using the light sensor object

lightsensor→get_friendlyName() lightsensor→friendlyName() lightsensor.get_friendlyName() YLightSensor

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

```
function get_friendlyName( )
```

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

Returns :

a string that uniquely identifies the light sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

lightsensor→get_functionDescriptor() lightsensor→functionDescriptor() lightsensor.get_functionDescriptor() YLightSensor

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

```
function 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() lightsensor→functionId()lightsensor.get_functionId() YLightSensor

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

```
function 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()
lightsensor→hardwareId()
lightsensor.get_hardwareId() YLightSensor

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

function **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() lightsensor→highestValue() lightsensor.get_highestValue() YLightSensor

Returns the maximal value observed for the ambiant light.

```
function get_highestValue( )
```

Returns :
a floating point number corresponding to the maximal value observed for the ambiant light

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

lightsensor→get_logFrequency()
lightsensor→logFrequency()
lightsensor.get_logFrequency() — YLightSensor

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

```
function get_logFrequency( )
```

Returns :

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

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

lightsensor→get_logicalName() lightsensor→logicalName() lightsensor.get_logicalName() YLightSensor

Returns the logical name of the light sensor.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the light sensor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

lightsensor→get_lowestValue()
lightsensor→lowestValue()
lightsensor.get_lowestValue()

YLightSensor

Returns the minimal value observed for the ambiant light.

```
function get_lowestValue( )
```

Returns :

a floating point number corresponding to the minimal value observed for the ambiant light

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

lightsensor→get_module() lightsensor→module()lightsensor.get_module() YLightSensor

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

```
function 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_module_async() lightsensor→module_async() lightsensor.get_module_async() — YLightSensor

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

lightsensor→get_recordedData() lightsensor→recordedData() lightsensor.get_recordedData() YLightSensor

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

```
function 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.

lightsensor→get_reportFrequency() lightsensor→reportFrequency() lightsensor.get_reportFrequency() YLightSensor

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

```
function 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() lightsensor→resolution()lightsensor.get_resolution() YLightSensor

Returns the resolution of the measured values.

```
function 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() lightsensor→unit()lightsensor.get_unit() YLightSensor

Returns the measuring unit for the ambiant light.

```
function get_unit( )
```

Returns :
a string corresponding to the measuring unit for the ambiant light

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

lightsensor→get_userData() lightsensor→userData()lightsensor.get_userData() YLightSensor

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

function **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.

```
function 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→isOnline_async() lightsensor.isOnline_async() YLightSensor

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

lightsensor→load()lightsensor.load() YLightSensor

Preloads the light sensor cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async()lightsensor.load_async() YLightSensor

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

lightsensor→nextLightSensor() lightsensor.nextLightSensor() YLightSensor

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

function **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.

```
function 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.

```
function registerValueCallback( callback)
```

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

Parameters :

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

lightsensor→set_highestValue()
lightsensor→setHighestValue()
lightsensor.set_highestValue() — YLightSensor

Changes the recorded maximal value observed for the ambiant light.

```
function set_highestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed for the ambiant light

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

lightsensor→set_logFrequency()
lightsensor→setLogFrequency()
lightsensor.set_logFrequency()

YLightSensor

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

lightsensor→set_logicalName() lightsensor→setLogicalName() lightsensor.set_logicalName()

YLightSensor

Changes the logical name of the light sensor.

```
function set_logicalName( newval)
```

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

Parameters :

newval a string corresponding to the logical name of the light sensor.

Returns :

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

lightsensor→set_lowestValue() lightsensor→setLowestValue() lightsensor.set_lowestValue()

YLightSensor

Changes the recorded minimal value observed for the ambiant light.

```
function set_lowestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed for the ambiant light

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

lightsensor→set_reportFrequency() lightsensor→setReportFrequency() lightsensor.set_reportFrequency() YLightSensor

Changes the timed value notification frequency for this function.

```
function 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.

```
function 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.

```
function 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

lightsensor→wait_async()lightsensor.wait_async() YLightSensor

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

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.23. Magnetometer function interface

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

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

js	<script type='text/javascript' src='yocto_magnetometer.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YMagnetometer = yoctolib.YMagnetometer;

php	require_once('yocto_magnetometer.php');
cpp	#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.

magnetometer→get_currentValue()

Returns the current value of the magnetic field.

magnetometer→get_errorMessage()

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

magnetometer→get_errorType()

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

magnetometer→get_friendlyName()

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

magnetometer→get_functionDescriptor()

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

magnetometer→get_functionId()

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

magnetometer→get_hardwareId()

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

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()yFindMagnetometer() YMagnetometer

Retrieves a magnetometer for a given identifier.

```
function yFindMagnetometer( 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()yFirstMagnetometer() YMagnetometer

Starts the enumeration of magnetometers currently accessible.

```
function yFirstMagnetometer( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

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

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

magnetometer→describe()magnetometer.describe() YMagnetometer

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

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

magnetometer→get_advertisedValue() magnetometer→advertisedValue() magnetometer.get_advertisedValue() YMagnetometer

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

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the magnetometer (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

magnetometer→get_currentRawValue() magnetometer→currentRawValue() magnetometer.get_currentRawValue() YMagnetometer

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

```
function get_currentRawValue( )
```

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

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

magnetometer→get_currentValue() magnetometer→currentValue() magnetometer.get_currentValue() YMagnetometer

Returns the current value of the magnetic field.

```
function get_currentValue( )
```

Returns :

a floating point number corresponding to the current value of the magnetic field

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

magnetometer→get_errorMessage() magnetometer→errorMessage() magnetometer.get_errorMessage() YMagnetometer

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

```
function get_errorMessage( )
```

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

Returns :

a string corresponding to the latest error message that occured while using the magnetometer object

magnetometer→get_errorType() magnetometer→errorType() magnetometer.get_errorType() YMagnetometer

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

```
function 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 occured while using the magnetometer object

magnetometer→get_friendlyName() magnetometer→friendlyName() magnetometer.get_friendlyName() YMagnetometer

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

```
function get_friendlyName( )
```

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

Returns :

a string that uniquely identifies the magnetometer using logical names (ex: `MyCustomName.relay1`)
On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

magnetometer→get_functionDescriptor() magnetometer→functionDescriptor() magnetometer.get_functionDescriptor() YMagnetometer

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

```
function get_functionDescriptor( )
```

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

Returns :

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

magnetometer→get_functionId() magnetometer→functionId() magnetometer.get_functionId() YMagnetometer

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

```
function get_functionId( )
```

For example `relay1`

Returns :

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

magnetometer→get_hardwareId() magnetometer→hardwareId() magnetometer.get_hardwareId() YMagnetometer

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

```
function 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()
magnetometer→highestValue()
magnetometer.get_highestValue()

YMagnetometer

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

```
function get_highestValue( )
```

Returns :

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

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

magnetometer→get_logFrequency() magnetometer→logFrequency() magnetometer.get_logFrequency() YMagnetometer

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

```
function get_logFrequency( )
```

Returns :

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

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

magnetometer→get_logicalName()
magnetometer→logicalName()
magnetometer.get_logicalName() YMagnetometer

Returns the logical name of the magnetometer.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the magnetometer. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

magnetometer→get_lowestValue() magnetometer→lowestValue() magnetometer.get_lowestValue()

YMagnetometer

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

```
function 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.

```
function 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_module_async() magnetometer→module_async() magnetometer.get_module_async() YMagnetometer

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

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

magnetometer→get_recordedData() magnetometer→recordedData() magnetometer.get_recordedData() YMagnetometer

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

```
function 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.

magnetometer→get_reportFrequency() magnetometer→reportFrequency() magnetometer.get_reportFrequency() YMagnetometer

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

```
function get_reportFrequency( )
```

Returns :

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

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

magnetometer→get_resolution() magnetometer→resolution() magnetometer.get_resolution() YMagnetometer

Returns the resolution of the measured values.

```
function 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() magnetometer→unit()magnetometer.get_unit() YMagnetometer

Returns the measuring unit for the magnetic field.

```
function get_unit( )
```

Returns :
a string corresponding to the measuring unit for the magnetic field

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

magnetometer→get_userData() magnetometer→userData() magnetometer.get_userData() YMagnetometer

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

```
function 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() magnetometer→xValue()magnetometer.get_xValue() YMagnetometer

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

```
function 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() magnetometer→yValue()magnetometer.get_yValue() YMagnetometer

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

```
function 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() magnetometer→zValue()magnetometer.get_zValue() YMagnetometer

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

```
function 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.

```
function 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→isOnline_async() magnetometer.isOnline_async() YMagnetometer

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

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

magnetometer→load()magnetometer.load() YMagnetometer

Preloads the magnetometer cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async() magnetometer.load_async() YMagnetometer

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

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

magnetometer→nextMagnetometer() magnetometer.nextMagnetometer() YMagnetometer

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

function **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.

```
function 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.

```
function registerValueCallback( callback)
```

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

Parameters :

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

magnetometer→set_highestValue() magnetometer→setHighestValue() magnetometer.set_highestValue()

YMagnetometer

Changes the recorded maximal value observed.

```
function 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.

```
function set_logFrequency( newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

magnetometer→set_logicalName() magnetometer→setLogicalName() magnetometer.set_logicalName() YMagnetometer

Changes the logical name of the magnetometer.

```
function set_logicalName( newval)
```

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

Parameters :

newval a string corresponding to the logical name of the magnetometer.

Returns :

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

magnetometer→set_lowestValue() magnetometer→setLowestValue() magnetometer.set_lowestValue() YMagnetometer

Changes the recorded minimal value observed.

```
function set_lowestValue( newval)
```

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

magnetometer→set_reportFrequency()
magnetometer→setReportFrequency()
magnetometer.set_reportFrequency()

YMagnetometer

Changes the timed value notification frequency for this function.

```
function 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.

```
function set_resolution( newval)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

magnetometer→set_userData()
magnetometer→setUserData()
magnetometer.set_userData()

YMagnetometer

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

```
function 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

magnetometer→wait_async() magnetometer.wait_async() YMagnetometer

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.24. Measured value

YMeasure objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the YDataSet class.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>

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()
measure→averageValue()
measure.get_averageValue()

YMeasure

Returns the average value observed during the time interval covered by this measure.

```
function get_averageValue( )
```

Returns :

a floating-point number corresponding to the average value observed.

measure→get_endTimeUTC() measure→endTimeUTC()measure.get_endTimeUTC() YMeasure

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

```
function get_endTimeUTC( )
```

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

Returns :
an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.

measure→get_maxValue() measure→maxValue()measure.get_maxValue() YMeasure

Returns the largest value observed during the time interval covered by this measure.

```
function get_maxValue( )
```

Returns :
a floating-point number corresponding to the largest value observed.

measure→get_minValue() measure→minValue()measure.get_minValue() YMeasure

Returns the smallest value observed during the time interval covered by this measure.

```
function get_minValue( )
```

Returns :
a floating-point number corresponding to the smallest value observed.

measure→get_startTimeUTC() measure→startTimeUTC() measure.get_startTimeUTC() YMeasure

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

```
function get_startTimeUTC( )
```

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

Returns :

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

3.25. Module control interface

This interface is identical for all Yoctopuce USB modules. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>

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→describe()
Returns a descriptive text that identifies the module.

module→download(pathname)
Downloads the specified built-in file and returns a binary buffer with its content.

module→functionCount()
Returns the number of functions (beside the "module" interface) available on the module.

module→functionId(functionIndex)
Retrieves the hardware identifier of the *n*th function on the module.

module→functionName(functionIndex)
Retrieves the logical name of the *n*th function on the module.

module→functionValue(functionIndex)
Retrieves the advertised value of the *n*th function on the module.

module→get_beacon()
Returns the state of the localization beacon.

module→get_errorMessage()
Returns the error message of the latest error with this module object.

module→get_errorType()
Returns the numerical error code of the latest error with this module object.

module→get_firmwareRelease()
Returns the version of the firmware embedded in the module.

module→get_hardwareId()
Returns the unique hardware identifier of the module.

module→get_icon2d()

Returns the icon of the module.

module→get_lastLogs()

Returns a string with last logs of the module.

module→get_logicalName()

Returns the logical name of the module.

module→get_luminosity()

Returns the luminosity of the module informative leds (from 0 to 100).

module→get_persistentSettings()

Returns the current state of persistent module settings.

module→get_productId()

Returns the USB device identifier of the module.

module→get_productName()

Returns the commercial name of the module, as set by the factory.

module→get_productRelease()

Returns the hardware release version of the module.

module→get_rebootCountdown()

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

module→get_serialNumber()

Returns the serial number of the module, as set by the factory.

module→get_upTime()

Returns the number of milliseconds spent since the module was powered on.

module→get_usbBandwidth()

Returns the number of USB interfaces used by the module.

module→get_usbCurrent()

Returns the current consumed by the module on the USB bus, in milli-amps.

module→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

module→isOnline()

Checks if the module is currently reachable, without raising any error.

module→isOnline_async(callback, context)

Checks if the module is currently reachable, without raising any error.

module→load(msValidity)

Preloads the module cache with a specified validity duration.

module→load_async(msValidity, callback, context)

Preloads the module cache with a specified validity duration (asynchronous version).

module→nextModule()

Continues the module enumeration started using `yFirstModule()`.

module→reboot(secBeforeReboot)

Schedules a simple module reboot after the given number of seconds.

module→registerLogCallback(callback)

todo

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_beacon(newval)

Turns on or off the module localization beacon.

module→set_logicalName(newval)

Changes the logical name of the module.

module→set_luminosity(newval)

Changes the luminosity of the module informative leds.

module→set_usbBandwidth(newval)

Changes the number of USB interfaces used by the module.

module→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

module→triggerFirmwareUpdate(secBeforeReboot)

Schedules a module reboot into special firmware update mode.

module→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YModule.FindModule() yFindModule()yFindModule()

YModule

Allows you to find a module from its serial number or from its logical name.

```
function yFindModule( 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()yFirstModule() YModule

Starts the enumeration of modules currently accessible.

```
function yFirstModule( )
```

Use the method `YModule.nextModule()` to iterate on the next modules.

Returns :

a pointer to a `YModule` object, corresponding to the first module currently online, or a `null` pointer if there are none.

module→describe()module.describe() YModule

Returns a descriptive text that identifies the module.

```
function 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.

```
function download( pathname)
```

Parameters :

pathname name of the new file to load

Returns :

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

module→functionCount()module.functionCount() YModule

Returns the number of functions (beside the "module" interface) available on the module.

```
function 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.

```
function 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.

```
function 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.

```
function functionValue( functionIndex)
```

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a short string (up to 6 characters) corresponding to the advertised value of the requested module function

On failure, throws an exception or returns an empty string.

module→get_beacon() module→beacon()module.get_beacon() YModule

Returns the state of the localization beacon.

```
function 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() module→errorMessage()module.get_errorMessage() YModule

Returns the error message of the latest error with this module object.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using this module object

module→get_errorType() module→errorType()module.get_errorType() YModule

Returns the numerical error code of the latest error with this module object.

```
function 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 occured while using this module object

module→get_firmwareRelease()
module→firmwareRelease()
module.get_firmwareRelease() YModule

Returns the version of the firmware embedded in the module.

```
function get_firmwareRelease( )
```

Returns :
a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns `Y_FIRMWARERELEASE_INVALID`.

module→get_hardwareId() module→hardwareId()module.get_hardwareId() YModule

Returns the unique hardware identifier of the module.

```
function 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() module→icon2d()module.get_icon2d() YModule

Returns the icon of the module.

```
function get_icon2d( )
```

The icon is a PNG image and does not exceeds 1536 bytes.

Returns :
a binary buffer with module icon, in png format.

module→get_lastLogs() module→lastLogs()module.get_lastLogs() YModule

Returns a string with last logs of the module.

```
function get_lastLogs( )
```

This method return only logs that are still in the module.

Returns :
a string with last logs of the module.

module→get_logicalName() module→logicalName()module.get_logicalName() YModule

Returns the logical name of the module.

```
function 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() module→luminosity()module.get_luminosity() YModule

Returns the luminosity of the module informative leds (from 0 to 100).

```
function get_luminosity( )
```

Returns :

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns `Y_LUMINOSITY_INVALID`.

module→get_persistentSettings() module→persistentSettings() module.get_persistentSettings() YModule

Returns the current state of persistent module settings.

```
function 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() module→productId()module.get_productId() YModule

Returns the USB device identifier of the module.

```
function 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() module→productName()module.get_productName() YModule

Returns the commercial name of the module, as set by the factory.

```
function get_productName( )
```

Returns :

a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns `Y_PRODUCTNAME_INVALID`.

module→get_productRelease() module→productRelease() module.get_productRelease() YModule

Returns the hardware release version of the module.

```
function get_productRelease( )
```

Returns :

an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns `Y_PRODUCTRELEASE_INVALID`.

module→get_rebootCountdown() module→rebootCountdown() module.get_rebootCountdown() YModule

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

```
function 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() module→serialNumber()module.get_serialNumber() YModule

Returns the serial number of the module, as set by the factory.

```
function 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() module→upTime()module.get_upTime() YModule

Returns the number of milliseconds spent since the module was powered on.

```
function get_upTime( )
```

Returns :

an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns `Y_UPTIME_INVALID`.

module→get_usbBandwidth() module→usbBandwidth() module.get_usbBandwidth() YModule

Returns the number of USB interfaces used by the module.

```
function get_usbBandwidth( )
```

Returns :

either `Y_USBBANDWIDTH_SIMPLE` or `Y_USBBANDWIDTH_DOUBLE`, according to the number of USB interfaces used by the module

On failure, throws an exception or returns `Y_USBBANDWIDTH_INVALID`.

module→get_usbCurrent() module→usbCurrent()module.get_usbCurrent() YModule

Returns the current consumed by the module on the USB bus, in milli-amps.

```
function get_usbCurrent( )
```

Returns :
an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns `Y_USBCURRENT_INVALID`.

module→get_userData() module→userData()module.get_userData() YModule

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

module→isOnline()module.isOnline() YModule

Checks if the module is currently reachable, without raising any error.

```
function 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→isOnline_async()module.isOnline_async() YModule

Checks if the module is currently reachable, without raising any error.

```
function isOnline_async( callback, context)
```

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.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→load()module.load() YModule

Preloads the module cache with a specified validity duration.

```
function 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 trafic 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→load_async()module.load_async() YModule

Preloads the module cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

msValidity	an integer corresponding to the validity of the loaded module parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→nextModule()module.nextModule() YModule

Continues the module enumeration started using `yFirstModule()`.

```
function 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.

```
function 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→revertFromFlash() module.revertFromFlash() YModule

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

function **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.

```
function saveToFlash( )
```

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

Returns :
`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

module→set_beacon() module→setBeacon()module.set_beacon() YModule

Turns on or off the module localization beacon.

```
function 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() module→setLogicalName()module.set_logicalName() YModule

Changes the logical name of the module.

```
function 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() module→setLuminosity()module.set_luminosity() YModule

Changes the luminosity of the module informative leds.

```
function set_luminosity( newval)
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the luminosity of the module informative leds

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_usbBandwidth() module→setUsbBandwidth() module.set_usbBandwidth() YModule

Changes the number of USB interfaces used by the module.

```
function set_usbBandwidth( newval)
```

You must reboot the module after changing this setting.

Parameters :

newval either `Y_USBBANDWIDTH_SIMPLE` or `Y_USBBANDWIDTH_DOUBLE`, according to the number of USB interfaces used by the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_userData() module→setUserData()module.set_userData() YModule

Stores a user context provided as argument in the userData attribute of the function.

```
function set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

module→triggerFirmwareUpdate() module.triggerFirmwareUpdate() YModule

Schedules a module reboot into special firmware update mode.

```
function 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→wait_async()module.wait_async() YModule

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.26. Network function interface

YNetwork objects provide access to TCP/IP parameters of Yoctopuce modules that include a built-in network interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_network.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YNetwork = yoctolib.YNetwork;

php	require_once('yocto_network.php');
cpp	#include "yocto_network.h"
m	#import "yocto_network.h"
pas	uses yocto_network;
vb	yocto_network.vb
cs	yocto_network.cs
java	import com.yoctopuce.YoctoAPI.YNetwork;
py	from yocto_network import *

Global functions

yFindNetwork(func)

Retrieves a network interface for a given identifier.

yFirstNetwork()

Starts the enumeration of network interfaces currently accessible.

`YNetwork` methods

network→callbackLogin(username, password)

Connects to the notification callback and saves the credentials required to log into it.

network→describe()

Returns a short text that describes unambiguously the instance of the network interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

network→get_adminPassword()

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

network→get_advertisedValue()

Returns the current value of the network interface (no more than 6 characters).

network→get_callbackCredentials()

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

network→get_callbackEncoding()

Returns the encoding standard to use for representing notification values.

network→get_callbackMaxDelay()

Returns the maximum waiting time between two callback notifications, in seconds.

network→get_callbackMethod()

Returns the HTTP method used to notify callbacks for significant state changes.

network→get_callbackMinDelay()

Returns the minimum waiting time between two callback notifications, in seconds.

network→get_callbackUrl()

Returns the callback URL to notify of significant state changes.

network→get_discoverable()

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

network→get_errorMessage()
Returns the error message of the latest error with the network interface.

network→get_errorType()
Returns the numerical error code of the latest error with the network interface.

network→get_friendlyName()
Returns a global identifier of the network interface in the format `MODULE_NAME.FUNCTION_NAME`.

network→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

network→get_functionId()
Returns the hardware identifier of the network interface, without reference to the module.

network→get_hardwareId()
Returns the unique hardware identifier of the network interface in the form `SERIAL.FUNCTIONID`.

network→get_ipAddress()
Returns the IP address currently in use by the device.

network→get_logicalName()
Returns the logical name of the network interface.

network→get_macAddress()
Returns the MAC address of the network interface.

network→get_module()
Gets the `YModule` object for the device on which the function is located.

network→get_module_async(callback, context)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

network→get_poeCurrent()
Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

network→get_primaryDNS()
Returns the IP address of the primary name server to be used by the module.

network→get_readiness()
Returns the current established working mode of the network interface.

network→get_router()
Returns the IP address of the router on the device subnet (default gateway).

network→get_secondaryDNS()
Returns the IP address of the secondary name server to be used by the module.

network→get_subnetMask()
Returns the subnet mask currently used by the device.

network→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

network→get_userPassword()
Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

network→get_wwwWatchdogDelay()
Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

network→isOnline()
Checks if the network interface is currently reachable, without raising any error.

network→isOnline_async(callback, context)
Checks if the network interface is currently reachable, without raising any error (asynchronous version).

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)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YNetwork.FindNetwork() yFindNetwork()yFindNetwork() YNetwork

Retrieves a network interface for a given identifier.

```
function yFindNetwork( 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() yFirstNetwork()yFirstNetwork() YNetwork

Starts the enumeration of network interfaces currently accessible.

```
function yFirstNetwork( )
```

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.

```
function 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`.

```
function 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() network→adminPassword() network.get_adminPassword() YNetwork

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

```
function get_adminPassword( )
```

Returns :

a string corresponding to a hash string if a password has been set for user "admin", or an empty string otherwise

On failure, throws an exception or returns `Y_ADMINPASSWORD_INVALID`.

network→get_advertisedValue() network→advertisedValue() network.get_advertisedValue() YNetwork

Returns the current value of the network interface (no more than 6 characters).

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the network interface (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

network→get_callbackCredentials() network→callbackCredentials() network.get_callbackCredentials()

YNetwork

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

```
function get_callbackCredentials( )
```

Returns :

a string corresponding to a hashed version of the notification callback credentials if set, or an empty string otherwise

On failure, throws an exception or returns `Y_CALLBACKCREDENTIALS_INVALID`.

network→get_callbackEncoding() network→callbackEncoding() network.get_callbackEncoding()

YNetwork

Returns the encoding standard to use for representing notification values.

```
function get_callbackEncoding( )
```

Returns :

a value among `Y_CALLBACKENCODING_FORM`, `Y_CALLBACKENCODING_JSON`, `Y_CALLBACKENCODING_JSON_ARRAY`, `Y_CALLBACKENCODING_CSV` and `Y_CALLBACKENCODING_YOCTO_API` corresponding to the encoding standard to use for representing notification values

On failure, throws an exception or returns `Y_CALLBACKENCODING_INVALID`.

network→get_callbackMaxDelay() network→callbackMaxDelay() network.get_callbackMaxDelay() YNetwork

Returns the maximum waiting time between two callback notifications, in seconds.

```
function get_callbackMaxDelay( )
```

Returns :

an integer corresponding to the maximum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns `Y_CALLBACKMAXDELAY_INVALID`.

network→get_callbackMethod()
network→callbackMethod()
network.get_callbackMethod() YNetwork

Returns the HTTP method used to notify callbacks for significant state changes.

```
function get_callbackMethod( )
```

Returns :

a value among `Y_CALLBACKMETHOD_POST`, `Y_CALLBACKMETHOD_GET` and `Y_CALLBACKMETHOD_PUT` corresponding to the HTTP method used to notify callbacks for significant state changes

On failure, throws an exception or returns `Y_CALLBACKMETHOD_INVALID`.

network→get_callbackMinDelay() network→callbackMinDelay() network.get_callbackMinDelay() YNetwork

Returns the minimum waiting time between two callback notifications, in seconds.

```
function 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() network→callbackUrl()network.get_callbackUrl() YNetwork

Returns the callback URL to notify of significant state changes.

```
function 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() network→discoverable()network.get_discoverable() YNetwork

Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

```
function get_discoverable( )
```

Returns :

either `Y_DISCOVERABLE_FALSE` or `Y_DISCOVERABLE_TRUE`, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

On failure, throws an exception or returns `Y_DISCOVERABLE_INVALID`.

network→get_errorMessage()
network→errorMessage()
network.get_errorMessage() YNetwork

Returns the error message of the latest error with the network interface.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occured while using the network interface object

network→get_errorType() network→errorType()network.get_errorType() YNetwork

Returns the numerical error code of the latest error with the network interface.

```
function 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 occured while using the network interface object

network→get_friendlyName() network→friendlyName()network.get_friendlyName() YNetwork

Returns a global identifier of the network interface in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the network interface if they are defined, otherwise the serial number of the module and the hardware identifier of the network interface (for exemple: `MyCustomName.relay1`)

Returns :
a string that uniquely identifies the network interface using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

network→get_functionDescriptor() network→functionDescriptor() network.get_functionDescriptor() YNetwork

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() network→functionId()network.get_functionId() YNetwork

Returns the hardware identifier of the network interface, without reference to the module.

```
function 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() network→hardwareId()network.get_hardwareId() YNetwork

Returns the unique hardware identifier of the network interface in the form `SERIAL.FUNCTIONID`.

```
function 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() network→ipAddress()network.get_ipAddress() YNetwork

Returns the IP address currently in use by the device.

```
function 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() network→logicalName()network.get_logicalName() YNetwork

Returns the logical name of the network interface.

```
function 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() network→macAddress()network.get_macAddress() YNetwork

Returns the MAC address of the network interface.

```
function get_macAddress( )
```

The MAC address is also available on a sticker on the module, in both numeric and barcode forms.

Returns :
a string corresponding to the MAC address of the network interface

On failure, throws an exception or returns `Y_MACADDRESS_INVALID`.

network→get_module() network→module()network.get_module() YNetwork

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() network→module_async() network.get_module_async() YNetwork

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

network→get_poeCurrent() network→poeCurrent()network.get_poeCurrent() YNetwork

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

```
function 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() network→primaryDNS()network.get_primaryDNS() YNetwork

Returns the IP address of the primary name server to be used by the module.

```
function 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() network→readiness()network.get_readiness() YNetwork

Returns the current established working mode of the network interface.

```
function 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() network→router()network.get_router() YNetwork

Returns the IP address of the router on the device subnet (default gateway).

```
function get_router( )
```

Returns :

a string corresponding to the IP address of the router on the device subnet (default gateway)

On failure, throws an exception or returns `Y_ROUTER_INVALID`.

network→get_secondaryDNS() network→secondaryDNS() network.get_secondaryDNS() YNetwork

Returns the IP address of the secondary name server to be used by the module.

```
function 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() network→subnetMask()network.get_subnetMask() YNetwork

Returns the subnet mask currently used by the device.

```
function 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() network→userData()network.get_userData() YNetwork

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

network→get_userPassword()
network→userPassword()
network.get_userPassword() YNetwork

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

```
function get_userPassword( )
```

Returns :

a string corresponding to a hash string if a password has been set for "user" user, or an empty string otherwise

On failure, throws an exception or returns `Y_USERPASSWORD_INVALID`.

network→get_wwwWatchdogDelay() network→wwwWatchdogDelay() network.get_wwwWatchdogDelay() YNetwork

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

```
function 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.

```
function isOnline( )
```

If there is a cached value for the network interface in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the network interface.

Returns :

`true` if the network interface can be reached, and `false` otherwise

network→isOnline_async()network.isOnline_async() YNetwork

Checks if the network interface is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

network→load()network.load() YNetwork

Preloads the network interface cache with a specified validity duration.

```
function 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 trafic 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→load_async()network.load_async() YNetwork

Preloads the network interface cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

network→nextNetwork()network.nextNetwork() YNetwork

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

```
function 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.

```
function 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.

```
function 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.

```
function set_adminPassword( newval)
```

This password becomes instantly required to perform any change of the module state. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the password for the "admin" user

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackCredentials() network→setCallbackCredentials() network.set_callbackCredentials() YNetwork

Changes the credentials required to connect to the callback address.

```
function 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 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.

```
function set_callbackEncoding( newval)
```

Parameters :

newval a value among `Y_CALLBACKENCODING_FORM`, `Y_CALLBACKENCODING_JSON`, `Y_CALLBACKENCODING_JSON_ARRAY`, `Y_CALLBACKENCODING_CSV` and `Y_CALLBACKENCODING_YOCTO_API` corresponding to the encoding standard to use for representing notification values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMaxDelay() network→setCallbackMaxDelay() network.set_callbackMaxDelay() YNetwork

Changes the maximum waiting time between two callback notifications, in seconds.

```
function set_callbackMaxDelay( newval)
```

Parameters :

newval an integer corresponding to the maximum waiting time between two callback notifications, in seconds

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMethod() network→setCallbackMethod() network.set_callbackMethod() YNetwork

Changes the HTTP method used to notify callbacks for significant state changes.

```
function set_callbackMethod( newval)
```

Parameters :

newval a value among `Y_CALLBACKMETHOD_POST`, `Y_CALLBACKMETHOD_GET` and `Y_CALLBACKMETHOD_PUT` corresponding to the HTTP method used to notify callbacks for significant state changes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMinDelay() network→setCallbackMinDelay() network.set_callbackMinDelay() YNetwork

Changes the minimum waiting time between two callback notifications, in seconds.

```
function 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() network→setCallbackUrl()network.set_callbackUrl() YNetwork

Changes the callback URL to notify significant state changes.

```
function set_callbackUrl( newval)
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the callback URL to notify significant state changes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_discoverable() network→setDiscoverable() network.set_discoverable() YNetwork

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

```
function set_discoverable( newval)
```

Parameters :

newval either `Y_DISCOVERABLE_FALSE` or `Y_DISCOVERABLE_TRUE`, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_logicalName() network→setLogicalName() network.set_logicalName() YNetwork

Changes the logical name of the network interface.

```
function 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() network→setPrimaryDNS()network.set_primaryDNS() YNetwork

Changes the IP address of the primary name server to be used by the module.

```
function set_primaryDNS( newval)
```

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval a string corresponding to the IP address of the primary name server to be used by the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_secondaryDNS() network→setSecondaryDNS() network.set_secondaryDNS() YNetwork

Changes the IP address of the secondary name server to be used by the module.

```
function 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() network→setUserData()network.set_userData() YNetwork

Stores a user context provided as argument in the userData attribute of the function.

```
function 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.

```
function set_userPassword( newval)
```

This password becomes instantly required to perform any use of the module. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the password for the "user" user

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_wwwWatchdogDelay() network→setWwwWatchdogDelay() network.set_wwwWatchdogDelay() YNetwork

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

```
function 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.

```
function 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 :

fallbackIpAddr	fallback IP address, to be used when no DHCP reply is received
fallbackSubnetMaskLen	fallback subnet mask length when no DHCP reply is received, as an integer (eg. 24 means 255.255.255.0)
fallbackRouter	fallback router IP address, to be used when no DHCP reply is received

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→useStaticIP()network.useStaticIP() YNetwork

Changes the configuration of the network interface to use a static IP address.

function **useStaticIP**(**ipAddress**, **subnetMaskLen**, **router**)

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ipAddress	device IP address
subnetMaskLen	subnet mask length, as an integer (eg. 24 means 255.255.255.0)
router	router IP address (default gateway)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→wait_async()network.wait_async() YNetwork

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.27. OS control

The OScontrol object allows some control over the operating system running a VirtualHub. OsControl is available on the VirtualHub software only. This feature must be activated at the VirtualHub start up with -o option.

In order to use the functions described here, you should include:

`js`	<script type='text/javascript' src='yocto_oscontrol.js'></script>

`nodejs`	var yoctolib = require('yoctolib');
	var YOsControl = yoctolib.YOsControl;

`php`	require_once('yocto_oscontrol.php');
`cpp`	#include "yocto_oscontrol.h"
`m`	#import "yocto_oscontrol.h"
`pas`	uses yocto_oscontrol;
`vb`	yocto_oscontrol.vb
`cs`	yocto_oscontrol.cs
`java`	import com.yoctopuce.YoctoAPI.YOsControl;
`py`	from yocto_oscontrol import *

Global functions

yFindOsControl(**func**)

Retrieves OS control for a given identifier.

yFirstOsControl()

Starts the enumeration of OS control currently accessible.

`YOsControl` methods

oscontrol→describe()

Returns a short text that describes unambiguously the instance of the OS control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

oscontrol→get_advertisedValue()

Returns the current value of the OS control (no more than 6 characters).

oscontrol→get_errorMessage()

Returns the error message of the latest error with the OS control.

oscontrol→get_errorType()

Returns the numerical error code of the latest error with the OS control.

oscontrol→get_friendlyName()

Returns a global identifier of the OS control in the format `MODULE_NAME.FUNCTION_NAME`.

oscontrol→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

oscontrol→get_functionId()

Returns the hardware identifier of the OS control, without reference to the module.

oscontrol→get_hardwareId()

Returns the unique hardware identifier of the OS control in the form `SERIAL.FUNCTIONID`.

oscontrol→get_logicalName()

Returns the logical name of the OS control.

oscontrol→get_module()

Gets the `YModule` object for the device on which the function is located.

oscontrol→get_module_async(**callback**, **context**)

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() yFindOsControl()yFindOsControl() YOsControl

Retrieves OS control for a given identifier.

```
function yFindOsControl( 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()yFirstOsControl() YOsControl

Starts the enumeration of OS control currently accessible.

```
function yFirstOsControl( )
```

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`.

```
function 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() oscontrol→advertisedValue() oscontrol.get_advertisedValue() YOsControl

Returns the current value of the OS control (no more than 6 characters).

```
function 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.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occured while using the OS control object

oscontrol→get_errorType() oscontrol→errorType()oscontrol.get_errorType() YOsControl

Returns the numerical error code of the latest error with the OS control.

```
function 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 occured while using the OS control object

oscontrol→get_friendlyName()
oscontrol→friendlyName()
oscontrol.get_friendlyName() — YOsControl

Returns a global identifier of the OS control in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the OS control if they are defined, otherwise the serial number of the module and the hardware identifier of the OS control (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the OS control using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

oscontrol→get_functionDescriptor() oscontrol→functionDescriptor() oscontrol.get_functionDescriptor() YOsControl

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() oscontrol→functionId()oscontrol.get_functionId() YOsControl

Returns the hardware identifier of the OS control, without reference to the module.

```
function 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() oscontrol→hardwareId()oscontrol.get_hardwareId() YOsControl

Returns the unique hardware identifier of the OS control in the form `SERIAL.FUNCTIONID`.

```
function 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()
oscontrol→logicalName()
oscontrol.get_logicalName()

YOsControl

Returns the logical name of the OS control.

```
function 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() oscontrol→module()oscontrol.get_module() YOsControl

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() oscontrol→module_async() oscontrol.get_module_async() YOsControl

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

oscontrol→get_shutdownCountdown() oscontrol→shutdownCountdown() oscontrol.get_shutdownCountdown() YOsControl

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

```
function 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() oscontrol→userData()oscontrol.get_userData() YOsControl

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function 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.

```
function 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→isOnline_async() oscontrol.isOnline_async() YOsControl

Checks if the OS control is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

oscontrol→load()oscontrol.load() **YOsControl**

Preloads the OS control cache with a specified validity duration.

```
function 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 trafic 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→load_async()oscontrol.load_async() YOsControl

Preloads the OS control cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

oscontrol→nextOsControl()
oscontrol.nextOsControl()

YOsControl

Continues the enumeration of OS control started using `yFirstOsControl()`.

```
function 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.

```
function registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

oscontrol→set_logicalName()
oscontrol→setLogicalName()
oscontrol.set_logicalName()

YOsControl

Changes the logical name of the OS control.

```
function 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() oscontrol→setUserData()oscontrol.set_userData() YOsControl

Stores a user context provided as argument in the userData attribute of the function.

```
function 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.

```
function 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.

oscontrol→wait_async()oscontrol.wait_async() YOsControl

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.28. Power function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_power.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YPower = yoctolib.YPower;

php	require_once('yocto_power.php');
cpp	#include "yocto_power.h"
m	#import "yocto_power.h"
pas	uses yocto_power;
vb	yocto_power.vb
cs	yocto_power.cs
java	import com.yoctopuce.YoctoAPI.YPower;
py	from yocto_power import *

Global functions

yFindPower(func)
Retrieves a electrical power sensor for a given identifier.

yFirstPower()
Starts the enumeration of electrical power sensors currently accessible.

`YPower` methods

power→calibrateFromPoints(rawValues, refValues)
Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

power→describe()
Returns a short text that describes unambiguously the instance of the electrical power sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

power→get_advertisedValue()
Returns the current value of the electrical power sensor (no more than 6 characters).

power→get_cosPhi()
Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

power→get_currentRawValue()
Returns the uncalibrated, unrounded raw value returned by the sensor.

power→get_currentValue()
Returns the current measure for the electrical power.

power→get_errorMessage()
Returns the error message of the latest error with the electrical power sensor.

power→get_errorType()
Returns the numerical error code of the latest error with the electrical power sensor.

power→get_friendlyName()
Returns a global identifier of the electrical power sensor in the format `MODULE_NAME.FUNCTION_NAME`.

power→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

power→get_functionId()

Returns the hardware identifier of the electrical power sensor, without reference to the module.

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.

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.

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 pour the electrical power.

power→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

power→set_logicalName(newval)

Changes the logical name of the electrical power sensor.

power→set_lowestValue(newval)

Changes the recorded minimal value observed pour the electrical power.

power→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

power→set_resolution(newval)

Changes the resolution of the measured values.

power→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

power→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPower.FindPower() yFindPower()yFindPower() — YPower

Retrieves a electrical power sensor for a given identifier.

```
function yFindPower( 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()yFirstPower()

YPower

Starts the enumeration of electrical power sensors currently accessible.

```
function yFirstPower( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues	array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
refValues	array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

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`.

```
function 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 electrical power sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

power→get_advertisedValue() power→advertisedValue() power.get_advertisedValue() YPower

Returns the current value of the electrical power sensor (no more than 6 characters).

function **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()
power→cosPhi()power.get_cosPhi() YPower

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

```
function get_cosPhi( )
```

Returns :

a floating point number corresponding to the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA)

On failure, throws an exception or returns `Y_COSPHI_INVALID`.

power→get_currentRawValue()
power→currentRawValue()
power.get_currentRawValue() — YPower

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( )
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

power→get_currentValue() power→currentValue()power.get_currentValue() YPower

Returns the current measure for the electrical power.

```
function get_currentValue( )
```

Returns :
a floating point number corresponding to the current measure for the electrical power

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

power→get_errorMessage() power→errorMessage()power.get_errorMessage() YPower

Returns the error message of the latest error with the electrical power sensor.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occured while using the electrical power sensor object

power→get_errorType() power→errorType()power.get_errorType() YPower

Returns the numerical error code of the latest error with the electrical power sensor.

```
function 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 occured while using the electrical power sensor object

power→get_friendlyName() power→friendlyName()power.get_friendlyName() YPower

Returns a global identifier of the electrical power sensor in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the electrical power sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the electrical power sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the electrical power sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

power→get_functionDescriptor() power→functionDescriptor() power.get_functionDescriptor() YPower

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() power→functionId()power.get_functionId() YPower

Returns the hardware identifier of the electrical power sensor, without reference to the module.

```
function 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() power→hardwareId()power.get_hardwareId() YPower

Returns the unique hardware identifier of the electrical power sensor in the form `SERIAL.FUNCTIONID`.

```
function 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() power→highestValue()power.get_highestValue() YPower

Returns the maximal value observed for the electrical power.

```
function get_highestValue( )
```

Returns :

a floating point number corresponding to the maximal value observed for the electrical power

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

power→get_logFrequency() power→logFrequency()power.get_logFrequency() YPower

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function 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() power→logicalName()power.get_logicalName() YPower

Returns the logical name of the electrical power sensor.

```
function 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() power→lowestValue()power.get_lowestValue() YPower

Returns the minimal value observed for the electrical power.

```
function get_lowestValue( )
```

Returns :

a floating point number corresponding to the minimal value observed for the electrical power

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

power→get_meter() power→meter()power.get_meter() YPower

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

```
function 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() power→meterTimer()power.get_meterTimer() YPower

Returns the elapsed time since last energy counter reset, in seconds.

```
function 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() power→module()power.get_module() YPower

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() power→module_async()power.get_module_async() YPower

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

power→get_recordedData() power→recordedData()power.get_recordedData() YPower

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime, endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime	the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.
endTime	the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

power→get_reportFrequency() power→reportFrequency() power.get_reportFrequency() YPower

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( )
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

power→get_resolution() power→resolution()power.get_resolution() YPower

Returns the resolution of the measured values.

```
function 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() power→unit()power.get_unit() YPower

Returns the measuring unit for the electrical power.

```
function 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() power→userData()power.get_userData() YPower

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function 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.

```
function 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→isOnline_async()power.isOnline_async() YPower

Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

power→load()power.load() YPower

Preloads the electrical power sensor cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async()power.load_async() YPower

Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

power→nextPower()power.nextPower() YPower

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

```
function 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.

function **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.

```
function 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.

```
function reset( )
```

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_highestValue() power→setHighestValue()power.set_highestValue() YPower

Changes the recorded maximal value observed pour the electrical power.

```
function set_highestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed pour the electrical power

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_logFrequency() power→setLogFrequency()power.set_logFrequency() YPower

Changes the datalogger recording frequency for this function.

```
function 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() power→setLogicalName()power.set_logicalName() YPower

Changes the logical name of the electrical power sensor.

function **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() power→setLowestValue()power.set_lowestValue() YPower

Changes the recorded minimal value observed pour the electrical power.

```
function set_lowestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed pour the electrical power

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_reportFrequency()
power→setReportFrequency()
power.set_reportFrequency()

YPower

Changes the timed value notification frequency for this function.

```
function 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() power→setResolution()power.set_resolution() YPower

Changes the resolution of the measured values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_userData() power→setUserData()power.set_userData() YPower

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

power→wait_async()power.wait_async() YPower

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.29. Pressure function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pressure.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YPressure = yoctolib.YPressure;

php	require_once('yocto_pressure.php');
cpp	#include "yocto_pressure.h"
m	#import "yocto_pressure.h"
pas	uses yocto_pressure;
vb	yocto_pressure.vb
cs	yocto_pressure.cs
java	import com.yoctopuce.YoctoAPI.YPressure;
py	from yocto_pressure import *

Global functions

yFindPressure(func)

Retrieves a pressure sensor for a given identifier.

yFirstPressure()

Starts the enumeration of pressure sensors currently accessible.

`YPressure` methods

pressure→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

pressure→describe()

Returns a short text that describes unambiguously the instance of the pressure sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

pressure→get_advertisedValue()

Returns the current value of the pressure sensor (no more than 6 characters).

pressure→get_currentRawValue()

Returns the unrounded and uncalibrated raw value returned by the sensor.

pressure→get_currentValue()

Returns the current measure for the pressure.

pressure→get_errorMessage()

Returns the error message of the latest error with the pressure sensor.

pressure→get_errorType()

Returns the numerical error code of the latest error with the pressure sensor.

pressure→get_friendlyName()

Returns a global identifier of the pressure sensor in the format `MODULE_NAME.FUNCTION_NAME`.

pressure→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

pressure→get_functionId()

Returns the hardware identifier of the pressure sensor, without reference to the module.

pressure→get_hardwareId()

Returns the unique hardware identifier of the pressure sensor in the form `SERIAL.FUNCTIONID`.

pressure→get_highestValue()

Returns the maximal value observed for the pressure.

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.

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 for the pressure.

pressure→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

pressure→set_logicalName(newval)

Changes the logical name of the pressure sensor.

pressure→set_lowestValue(newval)

Changes the recorded minimal value observed for the pressure.

pressure→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

pressure→set_resolution(newval)

Changes the resolution of the measured physical values.

pressure→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pressure→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPressure.FindPressure() yFindPressure()yFindPressure() YPressure

Retrieves a pressure sensor for a given identifier.

```
function yFindPressure( 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() yFirstPressure()yFirstPressure()

YPressure

Starts the enumeration of pressure sensors currently accessible.

```
function yFirstPressure( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues	array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
refValues	array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

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`.

```
function 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() pressure→advertisedValue() pressure.get_advertisedValue() YPressure

Returns the current value of the pressure sensor (no more than 6 characters).

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the pressure sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

pressure→get_currentRawValue() pressure→currentRawValue() pressure.get_currentRawValue()

YPressure

Returns the unrounded and uncalibrated raw value returned by the sensor.

```
function get_currentRawValue( )
```

Returns :

a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

pressure→get_currentValue() pressure→currentValue()pressure.get_currentValue() YPressure

Returns the current measure for the pressure.

```
function get_currentValue( )
```

Returns :

a floating point number corresponding to the current measure for the pressure

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

pressure→get_errorMessage() pressure→errorMessage() pressure.get_errorMessage() YPressure

Returns the error message of the latest error with the pressure sensor.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the pressure sensor object

pressure→get_errorType() pressure→errorType()pressure.get_errorType() YPressure

Returns the numerical error code of the latest error with the pressure sensor.

```
function 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 occured 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`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the pressure sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the pressure sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the pressure sensor using logical names (ex: `MyCustomName.relay1`)
On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

pressure→get_functionDescriptor() pressure→functionDescriptor() pressure.get_functionDescriptor() — YPressure

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() pressure→functionId()pressure.get_functionId() YPressure

Returns the hardware identifier of the pressure sensor, without reference to the module.

```
function 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() pressure→hardwareId()pressure.get_hardwareId() YPressure

Returns the unique hardware identifier of the pressure sensor in the form `SERIAL.FUNCTIONID`.

```
function 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.

```
function get_highestValue( )
```

Returns :

a floating point number corresponding to the maximal value observed for the pressure

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

pressure→get_logFrequency()
pressure→logFrequency()
pressure.get_logFrequency()

YPressure

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function 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() pressure→logicalName()pressure.get_logicalName() YPressure

Returns the logical name of the pressure sensor.

```
function 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() pressure→lowestValue()pressure.get_lowestValue() YPressure

Returns the minimal value observed for the pressure.

function **get_lowestValue**( )

Returns :
a floating point number corresponding to the minimal value observed for the pressure

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

pressure→get_module() pressure→module()pressure.get_module() YPressure

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() pressure→module_async() pressure.get_module_async() YPressure

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

pressure→get_recordedData()
pressure→recordedData()
pressure.get_recordedData() — YPressure

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function 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() pressure→reportFrequency() pressure.get_reportFrequency() — YPressure

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function 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() pressure→resolution()pressure.get_resolution() — YPressure

Returns the resolution of the measured values.

```
function 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() pressure→unit()pressure.get_unit() YPressure

Returns the measuring unit for the pressure.

```
function 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() pressure→userData()pressure.get_userData() YPressure

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function 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.

```
function 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→isOnline_async() pressure.isOnline_async() YPressure

Checks if the pressure sensor is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

pressure→load()pressure.load() YPressure

Preloads the pressure sensor cache with a specified validity duration.

```
function 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 trafic 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`.

function **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→load_async()pressure.load_async() YPressure

Preloads the pressure sensor cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

pressure→nextPressure()pressure.nextPressure() YPressure

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

```
function 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.

```
function 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.

```
function registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pressure→set_highestValue() pressure→setHighestValue() pressure.set_highestValue() YPressure

Changes the recorded maximal value observed for the pressure.

```
function set_highestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed for the pressure

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_logFrequency()
pressure→setLogFrequency()
pressure.set_logFrequency()

YPressure

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_logicalName() pressure→setLogicalName() pressure.set_logicalName() YPressure

Changes the logical name of the pressure sensor.

```
function set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the pressure sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

pressure→set_lowestValue() pressure→setLowestValue() pressure.set_lowestValue() YPressure

Changes the recorded minimal value observed for the pressure.

```
function set_lowestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed for the pressure

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_reportFrequency() pressure→setReportFrequency() pressure.set_reportFrequency() **YPressure**

Changes the timed value notification frequency for this function.

```
function 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() pressure→setResolution()pressure.set_resolution() YPressure

Changes the resolution of the measured physical values.

```
function 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() pressure→setUserData()pressure.set_userData() YPressure

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

pressure→wait_async()pressure.wait_async() YPressure

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.30. Pwm function interface

The Yoctopuce application programming interface allows you to configure, start, and stop the PWM.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pwmoutput.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YPwmOutput = yoctolib.YPwmOutput;

php	require_once('yocto_pwmoutput.php');
cpp	#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()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

pwmoutput→get_functionId()

Returns the hardware identifier of the PWM, without reference to the module.

pwmoutput→get_hardwareId()

Returns the unique hardware identifier of the PWM in the form `SERIAL.FUNCTIONID`.

pwmoutput→get_logicalName()

Returns the logical name of the PWM.

pwmoutput→get_module()

Gets the `YModule` object for the device on which the function is located.

pwmoutput→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

pwmoutput→get_period()

Returns the PWM period in milliseconds.

pwmoutput→get_pulseDuration()

Returns the PWM pulse length in milliseconds.

pwmoutput→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

pwmoutput→isOnline()

Checks if the PWM is currently reachable, without raising any error.

pwmoutput→isOnline_async(callback, context)

Checks if the PWM is currently reachable, without raising any error (asynchronous version).

pwmoutput→load(msValidity)

Preloads the PWM cache with a specified validity duration.

pwmoutput→load_async(msValidity, callback, context)

Preloads the PWM cache with a specified validity duration (asynchronous version).

pwmoutput→nextPwmOutput()

Continues the enumeration of PWMs started using `yFirstPwmOutput()`.

pwmoutput→pulseDurationMove(ms_target, ms_duration)

Performs a smooth transistion of the pulse duration toward a given value.

pwmoutput→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

pwmoutput→set_dutyCycle(newval)

Changes the PWM duty cycle, in per cents.

pwmoutput→set_dutyCycleAtPowerOn(newval)

Changes the PWM duty cycle at device power on.

pwmoutput→set_enabled(newval)

Stops or starts the PWM.

pwmoutput→set_enabledAtPowerOn(newval)

Changes the state of the PWM at device power on.

pwmoutput→set_frequency(newval)

Changes the PWM frequency.

pwmoutput→set_logicalName(newval)

Changes the logical name of the PWM.

pwmoutput→set_period(newval)

Changes the PWM period.

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()yFindPwmOutput() YPwmOutput

Retrieves a PWM for a given identifier.

```
function yFindPwmOutput( 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() yFirstPwmOutput()yFirstPwmOutput()

YPwmOutput

Starts the enumeration of PWMs currently accessible.

```
function yFirstPwmOutput( )
```

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`.

```
function 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.

```
function dutyCycleMove( target, ms_duration)
```

Parameters :

target	new duty cycle at the end of the transition (floating-point number, between 0 and 1)
ms_duration	total duration of the transition, in milliseconds

Returns :

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

pwmoutput→get_advertisedValue() pwmoutput→advertisedValue() pwmoutput.get_advertisedValue() YPwmOutput

Returns the current value of the PWM (no more than 6 characters).

function **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() pwmoutput→dutyCycle()pwmoutput.get_dutyCycle() YPwmOutput

Returns the PWM duty cycle, in per cents.

```
function get_dutyCycle( )
```

Returns :
a floating point number corresponding to the PWM duty cycle, in per cents

On failure, throws an exception or returns `Y_DUTYCYCLE_INVALID`.

pwmoutput→get_dutyCycleAtPowerOn() pwmoutput→dutyCycleAtPowerOn() pwmoutput.get_dutyCycleAtPowerOn() YPwmOutput

Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100

```
function 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() pwmoutput→enabled()pwmoutput.get_enabled() YPwmOutput

Returns the state of the PWMs.

```
function get_enabled( )
```

Returns :

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the state of the PWMs

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

pwmoutput→get_enabledAtPowerOn() pwmoutput→enabledAtPowerOn() pwmoutput.get_enabledAtPowerOn() YPwmOutput

Returns the state of the PWM at device power on.

```
function 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.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occured while using the PWM object

pwmoutput→get_errorType() pwmoutput→errorType()pwmoutput.get_errorType() YPwmOutput

Returns the numerical error code of the latest error with the PWM.

```
function 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 occured while using the PWM object

pwmoutput→get_frequency() pwmoutput→frequency()pwmoutput.get_frequency() YPwmOutput

Returns the PWM frequency in Hz.

```
function get_frequency( )
```

Returns :

an integer corresponding to the PWM frequency in Hz

On failure, throws an exception or returns `Y_FREQUENCY_INVALID`.

pwmoutput→get_friendlyName() pwmoutput→friendlyName() pwmoutput.get_friendlyName() — YPwmOutput

Returns a global identifier of the PWM in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the PWM if they are defined, otherwise the serial number of the module and the hardware identifier of the PWM (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the PWM using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

pwmoutput→get_functionDescriptor() pwmoutput→functionDescriptor() pwmoutput.get_functionDescriptor() YPwmOutput

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() pwmoutput→functionId()pwmoutput.get_functionId() YPwmOutput

Returns the hardware identifier of the PWM, without reference to the module.

```
function 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`.

function **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() pwmoutput→logicalName() pwmoutput.get_logicalName() YPwmOutput

Returns the logical name of the PWM.

```
function 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() pwmoutput→module()pwmoutput.get_module() YPwmOutput

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() pwmoutput→module_async() pwmoutput.get_module_async() YPwmOutput

Gets the `YModule` object for the device on which the function is located (asynchronous version).

function **get_module_async**(**callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

pwmoutput→get_period() pwmoutput→period()pwmoutput.get_period() YPwmOutput

Returns the PWM period in milliseconds.

```
function get_period( )
```

Returns :

a floating point number corresponding to the PWM period in milliseconds

On failure, throws an exception or returns `Y_PERIOD_INVALID`.

pwmoutput→get_pulseDuration()
pwmoutput→pulseDuration()
pwmoutput.get_pulseDuration() YPwmOutput

Returns the PWM pulse length in milliseconds.

```
function get_pulseDuration( )
```

Returns :

a floating point number corresponding to the PWM pulse length in milliseconds

On failure, throws an exception or returns `Y_PULSEDURATION_INVALID`.

pwmoutput→get_userData() pwmoutput→userData()pwmoutput.get_userData() YPwmOutput

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **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.

```
function 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→isOnline_async() pwmoutput.isOnline_async() YPwmOutput

Checks if the PWM is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

pwmoutput→load()pwmoutput.load() YPwmOutput

Preloads the PWM cache with a specified validity duration.

```
function 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 trafic 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→load_async()pwmoutput.load_async() YPwmOutput

Preloads the PWM cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

pwmoutput→nextPwmOutput() pwmoutput.nextPwmOutput() YPwmOutput

Continues the enumeration of PWMs started using `yFirstPwmOutput()`.

```
function 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.

```
function 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.

```
function registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pwmoutput→set_dutyCycle()
pwmoutput→setDutyCycle()
pwmoutput.set_dutyCycle()

YPwmOutput

Changes the PWM duty cycle, in per cents.

```
function 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.

```
function 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() pwmoutput→setEnabled()pwmoutput.set_enabled() YPwmOutput

Stops or starts the PWM.

```
function 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.

```
function 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.

```
function set_frequency( newval)
```

The duty cycle is kept unchanged thanks to an automatic pulse width change.

Parameters :

newval an integer corresponding to the PWM frequency

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_logicalName()
pwmoutput→setLogicalName()
pwmoutput.set_logicalName()

YPwmOutput

Changes the logical name of the PWM.

```
function 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() pwmoutput→setPeriod()pwmoutput.set_period() YPwmOutput

Changes the PWM period.

```
function set_period( newval)
```

Parameters :

newval a floating point number corresponding to the PWM period

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_pulseDuration()
pwmoutput→setPulseDuration()
pwmoutput.set_pulseDuration()

YPwmOutput

Changes the PWM pulse length, in milliseconds.

```
function 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.

```
function 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

pwmoutput→wait_async()pwmoutput.wait_async() YPwmOutput

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.31. PwmPowerSource function interface

The Yoctopuce application programming interface allows you to configure the voltage source used by all PWM on the same device.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_pwmpowersource.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YPwmPowerSource = yoctolib.YPwmPowerSource;

php	require_once('yocto_pwmpowersource.php');
cpp	#include "yocto_pwmpowersource.h"
m	#import "yocto_pwmpowersource.h"
pas	uses yocto_pwmpowersource;
vb	yocto_pwmpowersource.vb
cs	yocto_pwmpowersource.cs
java	import com.yoctopuce.YoctoAPI.YPwmPowerSource;
py	from yocto_pwmpowersource import *

Global functions

yFindPwmPowerSource(func)

Retrieves a voltage source for a given identifier.

yFirstPwmPowerSource()

Starts the enumeration of Voltage sources currently accessible.

`YPwmPowerSource` methods

pwmpowersource→describe()

Returns a short text that describes unambiguously the instance of the voltage source in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

pwmpowersource→get_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

pwmpowersource→get_errorMessage()

Returns the error message of the latest error with the voltage source.

pwmpowersource→get_errorType()

Returns the numerical error code of the latest error with the voltage source.

pwmpowersource→get_friendlyName()

Returns a global identifier of the voltage source in the format `MODULE_NAME.FUNCTION_NAME`.

pwmpowersource→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

pwmpowersource→get_functionId()

Returns the hardware identifier of the voltage source, without reference to the module.

pwmpowersource→get_hardwareId()

Returns the unique hardware identifier of the voltage source in the form `SERIAL.FUNCTIONID`.

pwmpowersource→get_logicalName()

Returns the logical name of the voltage source.

pwmpowersource→get_module()

Gets the `YModule` object for the device on which the function is located.

pwmpowersource→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

pwmpowersource→get_powerMode()
Returns the selected power source for the PWM on the same device

pwmpowersource→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

pwmpowersource→isOnline()
Checks if the voltage source is currently reachable, without raising any error.

pwmpowersource→isOnline_async(callback, context)
Checks if the voltage source is currently reachable, without raising any error (asynchronous version).

pwmpowersource→load(msValidity)
Preloads the voltage source cache with a specified validity duration.

pwmpowersource→load_async(msValidity, callback, context)
Preloads the voltage source cache with a specified validity duration (asynchronous version).

pwmpowersource→nextPwmPowerSource()
Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

pwmpowersource→registerValueCallback(callback)
Registers the callback function that is invoked on every change of advertised value.

pwmpowersource→set_logicalName(newval)
Changes the logical name of the voltage source.

pwmpowersource→set_powerMode(newval)
Changes the PWM power source.

pwmpowersource→set_userData(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()yFindPwmPowerSource() YPwmPowerSource

Retrieves a voltage source for a given identifier.

```
function yFindPwmPowerSource( 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()yFirstPwmPowerSource() YPwmPowerSource

Starts the enumeration of Voltage sources currently accessible.

```
function yFirstPwmPowerSource( )
```

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`.

```
function 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 source (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

pwmpowersource→get_advertisedValue() pwmpowersource→advertisedValue() pwmpowersource.get_advertisedValue() YPwmPowerSource

Returns the current value of the voltage source (no more than 6 characters).

function **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.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occured while using the voltage source object

pwmpowersource→get_errorType() pwmpowersource→errorType() pwmpowersource.get_errorType() YPwmPowerSource

Returns the numerical error code of the latest error with the voltage source.

```
function 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 occured while using the voltage source object

pwmpowersource→get_friendlyName() pwmpowersource→friendlyName() pwmpowersource.get_friendlyName() YPwmPowerSource

Returns a global identifier of the voltage source in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the voltage source if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage source (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the voltage source using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

pwmpowersource→get_functionDescriptor() pwmpowersource→functionDescriptor() pwmpowersource.get_functionDescriptor() YPwmPowerSource

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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.

```
function get_functionId( )
```

For example `relay1`

Returns :

a string that identifies the voltage source (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

pwmpowersource→get_hardwareId() pwmpowersource→hardwareId() pwmpowersource.get_hardwareId() YPwmPowerSource

Returns the unique hardware identifier of the voltage source in the form `SERIAL.FUNCTIONID`.

function **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() pwmpowersource→logicalName() pwmpowersource.get_logicalName() YPwmPowerSource

Returns the logical name of the voltage source.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the voltage source. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

pwmpowersource→get_module()
pwmpowersource→module()
pwmpowersource.get_module() YPwmPowerSource

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() pwmpowersource→module_async() pwmpowersource.get_module_async() YPwmPowerSource

Gets the `YModule` object for the device on which the function is located (asynchronous version).

function **get_module_async**(**callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

pwmpowersource→get_powerMode() pwmpowersource→powerMode() pwmpowersource.get_powerMode()

YPwmPowerSource

Returns the selected power source for the PWM on the same device

```
function get_powerMode( )
```

Returns :

a value among `Y_POWERMODE_USB_5V`, `Y_POWERMODE_USB_3V`, `Y_POWERMODE_EXT_V` and `Y_POWERMODE_OPNDRN` corresponding to the selected power source for the PWM on the same device

On failure, throws an exception or returns `Y_POWERMODE_INVALID`.

pwmpowersource→get_userData() pwmpowersource→userData() pwmpowersource.get_userData()

YPwmPowerSource

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

pwmpowersource→isOnline() pwmpowersource.isOnline() YPwmPowerSource

Checks if the voltage source is currently reachable, without raising any error.

```
function 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→isOnline_async() pwmpowersource.isOnline_async() YPwmPowerSource

Checks if the voltage source is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

pwmpowersource→load()pwmpowersource.load() YPwmPowerSource

Preloads the voltage source cache with a specified validity duration.

```
function 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 trafic 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→load_async() pwmpowersource.load_async()

YPwmPowerSource

Preloads the voltage source cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

pwmpowersource→nextPwmPowerSource() pwmpowersource.nextPwmPowerSource() YPwmPowerSource

Continues the enumeration of Voltage sources started using `yFirstPwmPowerSource()`.

```
function 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.

```
function registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback	the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

pwmpowersource→set_logicalName() pwmpowersource→setLogicalName() pwmpowersource.set_logicalName() YPwmPowerSource

Changes the logical name of the voltage source.

```
function 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.

```
function set_powerMode( newval)
```

PWM can use isolated 5V from USB, isolated 3V from USB or voltage from an external power source. The PWM can also work in open drain mode. In that mode, the PWM actively pulls the line down. Warning: this setting is common to all PWM on the same device. If you change that parameter, all PWM located on the same device are affected. If you want the change to be kept after a device reboot, make sure to call the matching module `saveToFlash()`.

Parameters :

newval a value among `Y_POWERMODE_USB_5V`, `Y_POWERMODE_USB_3V`, `Y_POWERMODE_EXT_V` and `Y_POWERMODE_OPNDRN` corresponding to the PWM power source

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmpowersource→set_userData() pwmpowersource→setUserData() pwmpowersource.set_userData()

YPwmPowerSource

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

pwmpowersource→wait_async() pwmpowersource.wait_async()

YPwmPowerSource

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.32. Quaternion interface

The Yoctopuce API YQt class provides direct access to the Yocto3D attitude estimation using a quaternion. It is usually not needed to use the YQt class directly, as the YGyro class provides a more convenient higher-level interface.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_gyro.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YGyro = yoctolib.YGyro;

php	require_once('yocto_gyro.php');
cpp	#include "yocto_gyro.h"
m	#import "yocto_gyro.h"
pas	uses yocto_gyro;
vb	yocto_gyro.vb
cs	yocto_gyro.cs
java	import com.yoctopuce.YoctoAPI.YGyro;
py	from yocto_gyro import *

Global functions
yFindQt(**func**) Retrieves a quaternion component for a given identifier.
yFirstQt() Starts the enumeration of quaternion components currently accessible.
YQt methods
qt→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
qt→describe() Returns a short text that describes unambiguously the instance of the quaternion component in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
qt→get_advertisedValue() Returns the current value of the quaternion component (no more than 6 characters).
qt→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor.
qt→get_currentValue() Returns the current value of the value.
qt→get_errorMessage() Returns the error message of the latest error with the quaternion component.
qt→get_errorType() Returns the numerical error code of the latest error with the quaternion component.
qt→get_friendlyName() Returns a global identifier of the quaternion component in the format `MODULE_NAME.FUNCTION_NAME`.
qt→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
qt→get_functionId() Returns the hardware identifier of the quaternion component, without reference to the module.
qt→get_hardwareId()

Returns the unique hardware identifier of the quaternion component in the form `SERIAL.FUNCTIONID`.

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)

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()yFindQt()

YQt

Retrieves a quaternion component for a given identifier.

```
function yFindQt( 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()yFirstQt()

YQt

Starts the enumeration of quaternion components currently accessible.

```
function yFirstQt( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues	array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
refValues	array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

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`.

```
function 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 quaternion component (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

qt→get_advertisedValue() qt→advertisedValue()qt.get_advertisedValue() YQt

Returns the current value of the quaternion component (no more than 6 characters).

```
function 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() qt→currentRawValue()qt.get_currentRawValue() YQt

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( )
```

Returns :
a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

qt→get_currentValue() qt→currentValue()qt.get_currentValue() YQt

Returns the current value of the value.

```
function get_currentValue( )
```

Returns :
a floating point number corresponding to the current value of the value

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

qt→get_errorMessage() qt→errorMessage()qt.get_errorMessage() YQt

Returns the error message of the latest error with the quaternion component.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the quaternion component object

qt→get_errorType()
qt→errorType()qt.get_errorType() **YQt**

Returns the numerical error code of the latest error with the quaternion component.

```
function 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 occured while using the quaternion component object

qt→get_friendlyName() qt→friendlyName()qt.get_friendlyName() **YQt**

Returns a global identifier of the quaternion component in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the quaternion component if they are defined, otherwise the serial number of the module and the hardware identifier of the quaternion component (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the quaternion component using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

qt→get_functionDescriptor() qt→functionDescriptor()qt.get_functionDescriptor() YQt

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() qt→functionId()qt.get_functionId() YQt

Returns the hardware identifier of the quaternion component, without reference to the module.

```
function 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() qt→hardwareId()qt.get_hardwareId() YQt

Returns the unique hardware identifier of the quaternion component in the form `SERIAL.FUNCTIONID`.

```
function 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() qt→highestValue()qt.get_highestValue() YQt

Returns the maximal value observed for the value since the device was started.

```
function get_highestValue( )
```

Returns :
a floating point number corresponding to the maximal value observed for the value since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

qt→get_logFrequency() qt→logFrequency()qt.get_logFrequency() YQt

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function 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() qt→logicalName()qt.get_logicalName() YQt

Returns the logical name of the quaternion component.

```
function get_logicalName( )
```

Returns :
a string corresponding to the logical name of the quaternion component. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

qt→get_lowestValue() qt→lowestValue()qt.get_lowestValue() YQt

Returns the minimal value observed for the value since the device was started.

```
function 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() qt→module()qt.get_module() **YQt**

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() qt→module_async()qt.get_module_async() YQt

Gets the `YModule` object for the device on which the function is located (asynchronous version).

function **get_module_async**(**callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

qt→get_recordedData() qt→recordedData()qt.get_recordedData() — YQt

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime, endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime	the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.
endTime	the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

qt→get_reportFrequency() qt→reportFrequency()qt.get_reportFrequency() YQt

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function 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() qt→resolution()qt.get_resolution() YQt

Returns the resolution of the measured values.

```
function get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

qt→get_unit()
qt→unit()qt.get_unit()

YQt

Returns the measuring unit for the value.

```
function get_unit( )
```

Returns :

a string corresponding to the measuring unit for the value

On failure, throws an exception or returns `Y_UNIT_INVALID`.

qt→get_userData() qt→userData()qt.get_userData() YQt

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function 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.

```
function 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→isOnline_async()qt.isOnline_async() YQt

Checks if the quaternion component is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

qt→load()qt.load() **YQt**

Preloads the quaternion component cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async()qt.load_async() YQt

Preloads the quaternion component cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

qt→nextQt()qt.nextQt() YQt

Continues the enumeration of quaternion components started using `yFirstQt()`.

function **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.

```
function 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.

```
function registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

qt→set_highestValue() qt→setHighestValue()qt.set_highestValue() YQt

Changes the recorded maximal value observed.

```
function 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() qt→setLogFrequency()qt.set_logFrequency() YQt

Changes the datalogger recording frequency for this function.

```
function 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() qt→setLogicalName()qt.set_logicalName() **YQt**

Changes the logical name of the quaternion component.

```
function set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the quaternion component.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

qt→set_lowestValue() qt→setLowestValue()qt.set_lowestValue() YQt

Changes the recorded minimal value observed.

```
function 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() qt→setReportFrequency()qt.set_reportFrequency() YQt

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_resolution() qt→setResolution()qt.set_resolution() YQt

Changes the resolution of the measured physical values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_userData() qt→setUserData()qt.set_userData() YQt

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

qt→wait_async()qt.wait_async() YQt

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.33. 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	<script type='text/javascript' src='yocto_realtimeclock.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YRealTimeClock = yoctolib.YRealTimeClock;

php	require_once('yocto_realtimeclock.php');
cpp	#include "yocto_realtimeclock.h"
m	#import "yocto_realtimeclock.h"
pas	uses yocto_realtimeclock;
vb	yocto_realtimeclock.vb
cs	yocto_realtimeclock.cs
java	import com.yoctopuce.YoctoAPI.YRealTimeClock;
py	from yocto_realtimeclock import *

Global functions

yFindRealTimeClock(func)

Retrieves a clock for a given identifier.

yFirstRealTimeClock()

Starts the enumeration of clocks currently accessible.

`YRealTimeClock` methods

realtimeclock→describe()

Returns a short text that describes unambiguously the instance of the clock in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

realtimeclock→get_advertisedValue()

Returns the current value of the clock (no more than 6 characters).

realtimeclock→get_dateTime()

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

realtimeclock→get_errorMessage()

Returns the error message of the latest error with the clock.

realtimeclock→get_errorType()

Returns the numerical error code of the latest error with the clock.

realtimeclock→get_friendlyName()

Returns a global identifier of the clock in the format `MODULE_NAME.FUNCTION_NAME`.

realtimeclock→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

realtimeclock→get_functionId()

Returns the hardware identifier of the clock, without reference to the module.

realtimeclock→get_hardwareId()

Returns the unique hardware identifier of the clock in the form `SERIAL.FUNCTIONID`.

realtimeclock→get_logicalName()

Returns the logical name of the clock.

realtimeclock→get_module()

Gets the `YModule` object for the device on which the function is located.

realtimeclock→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

realtimeclock→get_timeSet()

Returns true if the clock has been set, and false otherwise.

realtimeclock→get_unixTime()

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

realtimeclock→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

realtimeclock→get_utcOffset()

Returns the number of seconds between current time and UTC time (time zone).

realtimeclock→isOnline()

Checks if the clock is currently reachable, without raising any error.

realtimeclock→isOnline_async(callback, context)

Checks if the clock is currently reachable, without raising any error (asynchronous version).

realtimeclock→load(msValidity)

Preloads the clock cache with a specified validity duration.

realtimeclock→load_async(msValidity, callback, context)

Preloads the clock cache with a specified validity duration (asynchronous version).

realtimeclock→nextRealTimeClock()

Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

realtimeclock→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

realtimeclock→set_logicalName(newval)

Changes the logical name of the clock.

realtimeclock→set_unixTime(newval)

Changes the current time.

realtimeclock→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

realtimeclock→set_utcOffset(newval)

Changes the number of seconds between current time and UTC time (time zone).

realtimeclock→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRealTimeClock.FindRealTimeClock() yFindRealTimeClock()yFindRealTimeClock() YRealTimeClock

Retrieves a clock for a given identifier.

```
function yFindRealTimeClock( 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()yFirstRealTimeClock()

YRealTimeClock

Starts the enumeration of clocks currently accessible.

```
function yFirstRealTimeClock( )
```

Use the method `YRealTimeClock.nextRealTimeClock()` to iterate on next clocks.

Returns :

a pointer to a `YRealTimeClock` object, corresponding to the first clock currently online, or a `null` pointer if there are none.

realtimeclock→describe()realtimeclock.describe() YRealTimeClock

Returns a short text that describes unambiguously the instance of the clock in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function 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 clock (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

realtimeclock→get_advertisedValue() realtimeclock→advertisedValue() realtimeclock.get_advertisedValue() YRealTimeClock

Returns the current value of the clock (no more than 6 characters).

function **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() realtimeclock→dateTime() realtimeclock.get_dateTime() YRealTimeClock

Returns the current time in the form "YYYY/MM/DD hh:mm:ss"

```
function 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() realtimeclock→errorMessage() realtimeclock.get_errorMessage() YRealTimeClock

Returns the error message of the latest error with the clock.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the clock object

realtimeclock→get_errorType() realtimeclock→errorType() realtimeclock.get_errorType()

YRealTimeClock

Returns the numerical error code of the latest error with the clock.

```
function 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 occured while using the clock object

realtimeclock→get_friendlyName()
realtimeclock→friendlyName()
realtimeclock.get_friendlyName() YRealTimeClock

Returns a global identifier of the clock in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the clock if they are defined, otherwise the serial number of the module and the hardware identifier of the clock (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the clock using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

realtimeclock→get_functionDescriptor() realtimeclock→functionDescriptor() realtimeclock.get_functionDescriptor() YRealTimeClock

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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.

```
function get_functionId( )
```

For example `relay1`

Returns :

a string that identifies the clock (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

realtimeclock→get_hardwareId() realtimeclock→hardwareId() realtimeclock.get_hardwareId() YRealTimeClock

Returns the unique hardware identifier of the clock in the form `SERIAL.FUNCTIONID`.

```
function 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.

```
function 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() realtimeclock→module()realtimeclock.get_module() YRealTimeClock

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() realtimeclock→module_async() realtimeclock.get_module_async()

YRealTimeClock

Gets the `YModule` object for the device on which the function is located (asynchronous version).

function **get_module_async**(**callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

realtimeclock→get_timeSet() realtimeclock→timeSet()realtimeclock.get_timeSet() YRealTimeClock

Returns true if the clock has been set, and false otherwise.

```
function 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() realtimeclock→unixTime() realtimeclock.get_unixTime() YRealTimeClock

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

```
function get_unixTime( )
```

Returns :

an integer corresponding to the current time in Unix format (number of elapsed seconds since Jan 1st, 1970)

On failure, throws an exception or returns `Y_UNIXTIME_INVALID`.

realtimeclock→get_userData() realtimeclock→userData() realtimeclock.get_userData()

YRealTimeClock

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

realtimeclock→get_utcOffset() realtimeclock→utcOffset() realtimeclock.get_utcOffset() YRealTimeClock

Returns the number of seconds between current time and UTC time (time zone).

```
function 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.

```
function 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→isOnline_async() realtimeclock.isOnline_async()

YRealTimeClock

Checks if the clock is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

realtimeclock→load()realtimeclock.load() YRealTimeClock

Preloads the clock cache with a specified validity duration.

```
function 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 trafic 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→load_async() realtimeclock.load_async() YRealTimeClock

Preloads the clock cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

realtimeclock→nextRealTimeClock() realtimeclock.nextRealTimeClock() YRealTimeClock

Continues the enumeration of clocks started using `yFirstRealTimeClock()`.

function **nextRealTimeClock**( )

Returns :

a pointer to a `YRealTimeClock` object, corresponding to a clock currently online, or a `null` pointer if there are no more clocks to enumerate.

realtimeclock→registerValueCallback() realtimeclock.registerValueCallback() YRealTimeClock

Registers the callback function that is invoked on every change of advertised value.

```
function 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.

```
function 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.

```
function set_unixTime( newval)
```

Time is specifid 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.

function **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() realtimeclock→setUtcOffset() realtimeclock.set_utcOffset() YRealTimeClock

Changes the number of seconds between current time and UTC time (time zone).

```
function 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.

realtimeclock→wait_async() realtimeclock.wait_async() YRealTimeClock

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.34. Reference frame configuration

This class is used to setup the base orientation of the Yocto-3D, so that the orientation functions, relative to the earth surface plane, use the proper reference frame. The class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_refframe.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YRefFrame = yoctolib.YRefFrame;

php	require_once('yocto_refframe.php');
cpp	#include "yocto_refframe.h"
m	#import "yocto_refframe.h"
pas	uses yocto_refframe;
vb	yocto_refframe.vb
cs	yocto_refframe.cs
java	import com.yoctopuce.YoctoAPI.YRefFrame;
py	from yocto_refframe import *

Global functions

yFindRefFrame(func)

Retrieves a reference frame for a given identifier.

yFirstRefFrame()

Starts the enumeration of reference frames currently accessible.

`YRefFrame` methods

refframe→cancel3DCalibration()

Aborts the sensors tridimensional calibration process et restores normal settings.

refframe→describe()

Returns a short text that describes unambiguously the instance of the reference frame in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

refframe→get_3DCalibrationHint()

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

refframe→get_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.

refframe→get_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

refframe→get_3DCalibrationStage()

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

refframe→get_3DCalibrationStageProgress()

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

refframe→get_advertisedValue()

Returns the current value of the reference frame (no more than 6 characters).

refframe→get_bearing()

Returns the reference bearing used by the compass.

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)

Changes the logical name of the reference frame.

refframe→set_mountPosition(position, orientation)

Changes the compass and tilt sensor frame of reference.

refframe→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

refframe→start3DCalibration()

Initiates the sensors tridimensional calibration process.

refframe→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YRefFrame.FindRefFrame() yFindRefFrame()yFindRefFrame() — YRefFrame

Retrieves a reference frame for a given identifier.

```
function yFindRefFrame( 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() yFirstRefFrame()yFirstRefFrame() YRefFrame

Starts the enumeration of reference frames currently accessible.

```
function yFirstRefFrame( )
```

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.

```
function 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`.

```
function 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 reference frame (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

refframe→get_3DCalibrationHint() refframe→3DCalibrationHint() refframe.get_3DCalibrationHint() YRefFrame

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

```
function get_3DCalibrationHint( )
```

Returns :
a character string.

refframe→get_3DCalibrationLogMsg() refframe→3DCalibrationLogMsg() refframe.get_3DCalibrationLogMsg()

YRefFrame

Returns the latest log message from the calibration process.

```
function 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`.

```
function 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`.

```
function get_3DCalibrationStage( )
```

Returns :

an integer, growing each time a calibration stage is completed.

refframe→get_3DCalibrationStageProgress() refframe→3DCalibrationStageProgress() refframe.get_3DCalibrationStageProgress() YRefFrame

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

```
function get_3DCalibrationStageProgress( )
```

Returns :
an integer between 0 (not started) and 100 (stage completed).

refframe→get_advertisedValue() refframe→advertisedValue() refframe.get_advertisedValue() YRefFrame

Returns the current value of the reference frame (no more than 6 characters).

function **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() refframe→bearing()refframe.get_bearing() YRefFrame

Returns the reference bearing used by the compass.

```
function get_bearing( )
```

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here.

Returns :
a floating point number corresponding to the reference bearing used by the compass

On failure, throws an exception or returns `Y_BEARING_INVALID`.

refframe→get_errorMessage()
refframe→errorMessage()
refframe.get_errorMessage()

YRefFrame

Returns the error message of the latest error with the reference frame.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the reference frame object

refframe→get_errorType() refframe→errorType()refframe.get_errorType() YRefFrame

Returns the numerical error code of the latest error with the reference frame.

```
function 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 occured while using the reference frame object

refframe→get_friendlyName() refframe→friendlyName()refframe.get_friendlyName() YRefFrame

Returns a global identifier of the reference frame in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the reference frame if they are defined, otherwise the serial number of the module and the hardware identifier of the reference frame (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the reference frame using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

refframe→get_functionDescriptor()
refframe→functionDescriptor()
refframe.get_functionDescriptor() YRefFrame

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() refframe→functionId()refframe.get_functionId() YRefFrame

Returns the hardware identifier of the reference frame, without reference to the module.

```
function 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() refframe→hardwareId()refframe.get_hardwareId() YRefFrame

Returns the unique hardware identifier of the reference frame in the form `SERIAL.FUNCTIONID`.

function **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() refframe→logicalName()refframe.get_logicalName() YRefFrame

Returns the logical name of the reference frame.

```
function 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() refframe→module()refframe.get_module() YRefFrame

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() refframe→module_async() refframe.get_module_async()

YRefFrame

Gets the `YModule` object for the device on which the function is located (asynchronous version).

function **get_module_async**(**callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

refframe→get_mountOrientation() refframe→mountOrientation() refframe.get_mountOrientation() YRefFrame

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

```
function get_mountOrientation( )
```

Returns :

a value among the enumeration `Y_MOUNTORIENTATION` (`Y_MOUNTORIENTATION_TWELVE`, `Y_MOUNTORIENTATION_THREE`, `Y_MOUNTORIENTATION_SIX`, `Y_MOUNTORIENTATION_NINE`) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear.

On failure, throws an exception or returns a negative error code.

refframe→get_mountPosition()
refframe→mountPosition()
refframe.get_mountPosition() YRefFrame

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

```
function 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() refframe→userData()refframe.get_userData() YRefFrame

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function 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.

```
function 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→isOnline_async()refframe.isOnline_async() YRefFrame

Checks if the reference frame is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

refframe→load()refframe.load() YRefFrame

Preloads the reference frame cache with a specified validity duration.

```
function 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 trafic 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→load_async()refframe.load_async() YRefFrame

Preloads the reference frame cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

refframe→more3DCalibration() refframe.more3DCalibration() YRefFrame

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

```
function 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()`.

```
function 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.

```
function 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.

```
function 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() refframe→setBearing()refframe.set_bearing() **YRefFrame**

Changes the reference bearing used by the compass.

```
function set_bearing( newval)
```

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here. For instance, if you setup as reference bearing the value of the earth magnetic declination, the compass will provide the orientation relative to the geographic North. Similarly, when the sensor is not mounted along the standard directions because it has an additional yaw angle, you can set this angle in the reference bearing so that the compass provides the expected natural direction. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a floating point number corresponding to the reference bearing used by the compass

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

refframe→set_logicalName() refframe→setLogicalName() refframe.set_logicalName() YRefFrame

Changes the logical name of the reference frame.

```
function 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.

```
function 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() refframe→setUserData()refframe.set_userData() YRefFrame

Stores a user context provided as argument in the userData attribute of the function.

```
function 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.

```
function 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.

refframe→wait_async()refframe.wait_async() YRefFrame

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.35. Relay function interface

The Yoctopuce application programming interface allows you to switch the relay state. This change is not persistent: the relay will automatically return to its idle position whenever power is lost or if the module is restarted. The library can also generate automatically short pulses of determined duration. On devices with two output for each relay (double throw), the two outputs are named A and B, with output A corresponding to the idle position (at power off) and the output B corresponding to the active state. If you prefer the alternate default state, simply switch your cables on the board.

In order to use the functions described here, you should include:

js	`<script type='text/javascript' src='yocto_relay.js'></script>`

nodejs	var yoctolib = require('yoctolib');
	var YRelay = yoctolib.YRelay;

php	require_once('yocto_relay.php');
cpp	#include "yocto_relay.h"
m	#import "yocto_relay.h"
pas	uses yocto_relay;
vb	yocto_relay.vb
cs	yocto_relay.cs
java	import com.yoctopuce.YoctoAPI.YRelay;
py	from yocto_relay import *

Global functions

yFindRelay(func)
Retrieves a relay for a given identifier.

yFirstRelay()
Starts the enumeration of relays currently accessible.

`YRelay` methods

relay→delayedPulse(ms_delay, ms_duration)
Schedules a pulse.

relay→describe()
Returns a short text that describes unambiguously the instance of the relay in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

relay→get_advertisedValue()
Returns the current value of the relay (no more than 6 characters).

relay→get_countdown()
Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

relay→get_errorMessage()
Returns the error message of the latest error with the relay.

relay→get_errorType()
Returns the numerical error code of the latest error with the relay.

relay→get_friendlyName()
Returns a global identifier of the relay in the format `MODULE_NAME.FUNCTION_NAME`.

relay→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

relay→get_functionId()
Returns the hardware identifier of the relay, without reference to the module.

relay→get_hardwareId()

Returns the unique hardware identifier of the relay in the form `SERIAL.FUNCTIONID`.

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)

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()yFindRelay() YRelay

Retrieves a relay for a given identifier.

```
function yFindRelay( 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() yFirstRelay()yFirstRelay()

YRelay

Starts the enumeration of relays currently accessible.

```
function yFirstRelay( )
```

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.

```
function delayedPulse( ms_delay, ms_duration)
```

Parameters :

ms_delay	waiting time before the pulse, in millisecondes
ms_duration	pulse duration, in millisecondes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→describe()relay.describe() YRelay

Returns a short text that describes unambiguously the instance of the relay in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function 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 relay (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

relay→get_advertisedValue() relay→advertisedValue()relay.get_advertisedValue() YRelay

Returns the current value of the relay (no more than 6 characters).

```
function 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() relay→countdown()relay.get_countdown() **YRelay**

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

```
function 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() relay→errorMessage()relay.get_errorMessage() YRelay

Returns the error message of the latest error with the relay.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the relay object

relay→get_errorType() relay→errorType()relay.get_errorType() YRelay

Returns the numerical error code of the latest error with the relay.

```
function 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 occured while using the relay object

relay→get_friendlyName() relay→friendlyName()relay.get_friendlyName() YRelay

Returns a global identifier of the relay in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the relay if they are defined, otherwise the serial number of the module and the hardware identifier of the relay (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the relay using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

relay→get_functionDescriptor() relay→functionDescriptor() relay.get_functionDescriptor() YRelay

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() relay→functionId()relay.get_functionId() YRelay

Returns the hardware identifier of the relay, without reference to the module.

```
function 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() relay→hardwareId()relay.get_hardwareId() YRelay

Returns the unique hardware identifier of the relay in the form `SERIAL.FUNCTIONID`.

```
function 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() relay→logicalName()relay.get_logicalName() YRelay

Returns the logical name of the relay.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the relay. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

relay→get_maxTimeOnStateA() relay→maxTimeOnStateA() relay.get_maxTimeOnStateA() YRelay

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

```
function get_maxTimeOnStateA( )
```

Zero means no maximum time.

Returns :

an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEA_INVALID`.

relay→get_maxTimeOnStateB() relay→maxTimeOnStateB() relay.get_maxTimeOnStateB() YRelay

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

```
function get_maxTimeOnStateB( )
```

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEB_INVALID`.

relay→get_module() relay→module()relay.get_module() YRelay

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() relay→module_async()relay.get_module_async() YRelay

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

relay→get_output() relay→output()relay.get_output() YRelay

Returns the output state of the relays, when used as a simple switch (single throw).

```
function 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() relay→pulseTimer()relay.get_pulseTimer() YRelay

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

```
function 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() relay→state()relay.get_state() YRelay

Returns the state of the relays (A for the idle position, B for the active position).

```
function 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() relay→stateAtPowerOn()relay.get_stateAtPowerOn() YRelay

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

```
function 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() relay→userData()relay.get_userData() YRelay

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function 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.

```
function 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→isOnline_async()relay.isOnline_async() YRelay

Checks if the relay is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

relay→load()relay.load() YRelay

Preloads the relay cache with a specified validity duration.

```
function 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 trafic 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→load_async()relay.load_async() YRelay

Preloads the relay cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

relay→nextRelay()relay.nextRelay() YRelay

Continues the enumeration of relays started using `yFirstRelay()`.

```
function 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).

```
function pulse( ms_duration)
```

Parameters :

ms_duration pulse duration, in millisecondes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→registerValueCallback() relay.registerValueCallback() YRelay

Registers the callback function that is invoked on every change of advertised value.

```
function 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() relay→setLogicalName()relay.set_logicalName() YRelay

Changes the logical name of the relay.

```
function set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the relay.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

relay→set_maxTimeOnStateA() relay→setMaxTimeOnStateA() relay.set_maxTimeOnStateA() YRelay

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

```
function set_maxTimeOnStateA( newval)
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_maxTimeOnStateB() relay→setMaxTimeOnStateB() relay.set_maxTimeOnStateB() YRelay

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

```
function 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() relay→setOutput()relay.set_output() YRelay

Changes the output state of the relays, when used as a simple switch (single throw).

```
function 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() relay→setState()relay.set_state() YRelay

Changes the state of the relays (A for the idle position, B for the active position).

```
function set_state( newval)
```

Parameters :

newval either `Y_STATE_A` or `Y_STATE_B`, according to the state of the relays (A for the idle position, B for the active position)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_stateAtPowerOn() relay→setStateAtPowerOn() relay.set_stateAtPowerOn() YRelay

Preset the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

```
function 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() relay→setUserData()relay.set_userData() YRelay

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

relay→wait_async()relay.wait_async() YRelay

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.36. Sensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_api.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YAPI = yoctolib.YAPI;
	var YModule = yoctolib.YModule;

php	require_once('yocto_api.php');
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *

Global functions

yFindSensor(func)

Retrieves a sensor for a given identifier.

yFirstSensor()

Starts the enumeration of sensors currently accessible.

`YSensor` methods

sensor→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

sensor→describe()

Returns a short text that describes unambiguously the instance of the sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

sensor→get_advertisedValue()

Returns the current value of the sensor (no more than 6 characters).

sensor→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

sensor→get_currentValue()

Returns the current value of the measure.

sensor→get_errorMessage()

Returns the error message of the latest error with the sensor.

sensor→get_errorType()

Returns the numerical error code of the latest error with the sensor.

sensor→get_friendlyName()

Returns a global identifier of the sensor in the format `MODULE_NAME.FUNCTION_NAME`.

sensor→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

sensor→get_functionId()

Returns the hardware identifier of the sensor, without reference to the module.

sensor→get_hardwareId()

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)

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()yFindSensor() YSensor

Retrieves a sensor for a given identifier.

```
function yFindSensor( 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() yFirstSensor()yFirstSensor() YSensor

Starts the enumeration of sensors currently accessible.

```
function yFirstSensor( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues	array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
refValues	array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→describe()sensor.describe() YSensor

Returns a short text that describes unambiguously the instance of the sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function 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 sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

sensor→get_advertisedValue() sensor→advertisedValue() sensor.get_advertisedValue() YSensor

Returns the current value of the sensor (no more than 6 characters).

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

sensor→get_currentRawValue()
sensor→currentRawValue()
sensor.get_currentRawValue() YSensor

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( )
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

sensor→get_currentValue() sensor→currentValue()sensor.get_currentValue() YSensor

Returns the current value of the measure.

```
function get_currentValue( )
```

Returns :
a floating point number corresponding to the current value of the measure

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

sensor→get_errorMessage() sensor→errorMessage()sensor.get_errorMessage() YSensor

Returns the error message of the latest error with the sensor.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the sensor object

sensor→get_errorType() sensor→errorType()sensor.get_errorType() YSensor

Returns the numerical error code of the latest error with the sensor.

```
function 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 occured while using the sensor object

sensor→get_friendlyName() sensor→friendlyName()sensor.get_friendlyName() YSensor

Returns a global identifier of the sensor in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

sensor→get_functionDescriptor()
sensor→functionDescriptor()
sensor.get_functionDescriptor() YSensor

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() sensor→functionId()sensor.get_functionId() YSensor

Returns the hardware identifier of the sensor, without reference to the module.

```
function 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() sensor→hardwareId()sensor.get_hardwareId() YSensor

Returns the unique hardware identifier of the sensor in the form `SERIAL.FUNCTIONID`.

function **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() sensor→highestValue()sensor.get_highestValue() YSensor

Returns the maximal value observed for the measure since the device was started.

function **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() sensor→logFrequency()sensor.get_logFrequency() YSensor

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

function **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() sensor→logicalName()sensor.get_logicalName() YSensor

Returns the logical name of the sensor.

```
function 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() sensor→lowestValue()sensor.get_lowestValue() YSensor

Returns the minimal value observed for the measure since the device was started.

```
function 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() sensor→module()sensor.get_module() YSensor

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() sensor→module_async()sensor.get_module_async() YSensor

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

sensor→get_recordedData() sensor→recordedData()sensor.get_recordedData() YSensor

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime, endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime	the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.
endTime	the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

sensor→get_reportFrequency() sensor→reportFrequency() sensor.get_reportFrequency() YSensor

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( )
```

Returns :
a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

sensor→get_resolution() sensor→resolution()sensor.get_resolution() YSensor

Returns the resolution of the measured values.

```
function 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() sensor→unit()sensor.get_unit() YSensor

Returns the measuring unit for the measure.

```
function 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() sensor→userData()sensor.get_userData() YSensor

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function 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.

```
function 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→isOnline_async()sensor.isOnline_async() YSensor

Checks if the sensor is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

sensor→load()sensor.load() YSensor

Preloads the sensor cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async()sensor.load_async() YSensor

Preloads the sensor cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

sensor→nextSensor()sensor.nextSensor() YSensor

Continues the enumeration of sensors started using `yFirstSensor()`.

```
function 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.

function **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.

```
function 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() sensor→setHighestValue()sensor.set_highestValue() YSensor

Changes the recorded maximal value observed.

```
function set_highestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_logFrequency()
sensor→setLogFrequency()
sensor.set_logFrequency() YSensor

Changes the datalogger recording frequency for this function.

```
function 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() sensor→setLogicalName()sensor.set_logicalName() YSensor

Changes the logical name of the sensor.

```
function 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() sensor→setLowestValue()sensor.set_lowestValue() YSensor

Changes the recorded minimal value observed.

```
function set_lowestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_reportFrequency() sensor→setReportFrequency() sensor.set_reportFrequency() YSensor

Changes the timed value notification frequency for this function.

```
function 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() sensor→setResolution()sensor.set_resolution() **YSensor**

Changes the resolution of the measured physical values.

```
function 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() sensor→setUserData()sensor.set_userData() YSensor

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

sensor→wait_async()sensor.wait_async() YSensor

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.37. Servo function interface

Yoctopuce application programming interface allows you not only to move a servo to a given position, but also to specify the time interval in which the move should be performed. This makes it possible to synchronize two servos involved in a same move.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_servo.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YServo = yoctolib.YServo;

php	require_once('yocto_servo.php');
cpp	#include "yocto_servo.h"
m	#import "yocto_servo.h"
pas	uses yocto_servo;
vb	yocto_servo.vb
cs	yocto_servo.cs
java	import com.yoctopuce.YoctoAPI.YServo;
py	from yocto_servo import *

Global functions

yFindServo(**func**)

Retrieves a servo for a given identifier.

yFirstServo()

Starts the enumeration of servos currently accessible.

`YServo` methods

servo→describe()

Returns a short text that describes unambiguously the instance of the servo in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

servo→get_advertisedValue()

Returns the current value of the servo (no more than 6 characters).

servo→get_enabled()

Returns the state of the servos.

servo→get_enabledAtPowerOn()

Returns the servo signal generator state at power up.

servo→get_errorMessage()

Returns the error message of the latest error with the servo.

servo→get_errorType()

Returns the numerical error code of the latest error with the servo.

servo→get_friendlyName()

Returns a global identifier of the servo in the format `MODULE_NAME.FUNCTION_NAME`.

servo→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

servo→get_functionId()

Returns the hardware identifier of the servo, without reference to the module.

servo→get_hardwareId()

Returns the unique hardware identifier of the servo in the form `SERIAL.FUNCTIONID`.

servo→get_logicalName()

Returns the logical name of the servo.

servo→get_module()

Gets the `YModule` object for the device on which the function is located.

servo→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

servo→get_neutral()

Returns the duration in microseconds of a neutral pulse for the servo.

servo→get_position()

Returns the current servo position.

servo→get_positionAtPowerOn()

Returns the servo position at device power up.

servo→get_range()

Returns the current range of use of the servo.

servo→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

servo→isOnline()

Checks if the servo is currently reachable, without raising any error.

servo→isOnline_async(callback, context)

Checks if the servo is currently reachable, without raising any error (asynchronous version).

servo→load(msValidity)

Preloads the servo cache with a specified validity duration.

servo→load_async(msValidity, callback, context)

Preloads the servo cache with a specified validity duration (asynchronous version).

servo→move(target, ms_duration)

Performs a smooth move at constant speed toward a given position.

servo→nextServo()

Continues the enumeration of servos started using `yFirstServo()`.

servo→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

servo→set_enabled(newval)

Stops or starts the servo.

servo→set_enabledAtPowerOn(newval)

Configure the servo signal generator state at power up.

servo→set_logicalName(newval)

Changes the logical name of the servo.

servo→set_neutral(newval)

Changes the duration of the pulse corresponding to the neutral position of the servo.

servo→set_position(newval)

Changes immediately the servo driving position.

servo→set_positionAtPowerOn(newval)

Configure the servo position at device power up.

servo→set_range(newval)

Changes the range of use of the servo, specified in per cents.

servo→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

servo→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YServo.FindServo() yFindServo()yFindServo() YServo

Retrieves a servo for a given identifier.

```
function yFindServo( 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()yFirstServo() — YServo

Starts the enumeration of servos currently accessible.

```
function yFirstServo( )
```

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`.

```
function 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 servo (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

servo→get_advertisedValue() servo→advertisedValue()servo.get_advertisedValue() YServo

Returns the current value of the servo (no more than 6 characters).

```
function 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.

```
function get_enabled( )
```

Returns :

either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the state of the servos

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

servo→get_enabledAtPowerOn() servo→enabledAtPowerOn() servo.get_enabledAtPowerOn() YServo

Returns the servo signal generator state at power up.

```
function get_enabledAtPowerOn( )
```

Returns :

either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the servo signal generator state at power up

On failure, throws an exception or returns `Y_ENABLEDATPOWERON_INVALID`.

servo→get_errorMessage() servo→errorMessage()servo.get_errorMessage() YServo

Returns the error message of the latest error with the servo.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the servo object

servo→get_errorType() servo→errorType()servo.get_errorType() YServo

Returns the numerical error code of the latest error with the servo.

```
function 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 occured while using the servo object

servo→get_friendlyName() servo→friendlyName()servo.get_friendlyName() YServo

Returns a global identifier of the servo in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the servo if they are defined, otherwise the serial number of the module and the hardware identifier of the servo (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the servo using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

servo→get_functionDescriptor() servo→functionDescriptor() servo.get_functionDescriptor() YServo

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() servo→functionId()servo.get_functionId() YServo

Returns the hardware identifier of the servo, without reference to the module.

```
function 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() servo→hardwareId()servo.get_hardwareId() YServo

Returns the unique hardware identifier of the servo in the form `SERIAL.FUNCTIONID`.

```
function 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() servo→logicalName()servo.get_logicalName() YServo

Returns the logical name of the servo.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the servo. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

servo→get_module() servo→module()servo.get_module() YServo

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() servo→module_async()servo.get_module_async() YServo

Gets the `YModule` object for the device on which the function is located (asynchronous version).

function **get_module_async**(**callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

servo→get_neutral() servo→neutral()servo.get_neutral() YServo

Returns the duration in microseconds of a neutral pulse for the servo.

```
function get_neutral( )
```

Returns :

an integer corresponding to the duration in microseconds of a neutral pulse for the servo

On failure, throws an exception or returns `Y_NEUTRAL_INVALID`.

servo→get_position() servo→position()servo.get_position() YServo

Returns the current servo position.

```
function get_position( )
```

Returns :
an integer corresponding to the current servo position

On failure, throws an exception or returns `Y_POSITION_INVALID`.

servo→get_positionAtPowerOn() servo→positionAtPowerOn() servo.get_positionAtPowerOn() YServo

Returns the servo position at device power up.

```
function 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() servo→range()servo.get_range() YServo

Returns the current range of use of the servo.

```
function 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() servo→userData()servo.get_userData() YServo

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **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.

```
function 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→isOnline_async()servo.isOnline_async() YServo

Checks if the servo is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

servo→load()servo.load() YServo

Preloads the servo cache with a specified validity duration.

```
function 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 trafic 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→load_async()servo.load_async() YServo

Preloads the servo cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

servo→move()servo.move() YServo

Performs a smooth move at constant speed toward a given position.

```
function 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()`.

```
function 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.

```
function 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.

```
function 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.

```
function 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() servo→setLogicalName()servo.set_logicalName() YServo

Changes the logical name of the servo.

```
function set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the servo.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

servo→set_neutral() servo→setNeutral()servo.set_neutral() YServo

Changes the duration of the pulse corresponding to the neutral position of the servo.

```
function 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() servo→setPosition()servo.set_position() YServo

Changes immediately the servo driving position.

```
function 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.

```
function 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() servo→setRange()servo.set_range() YServo

Changes the range of use of the servo, specified in per cents.

```
function 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() servo→setUserData()servo.set_userData() YServo

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

servo→wait_async()servo.wait_async() YServo

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.38. Temperature function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_temperature.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YTemperature = yoctolib.YTemperature;

php	require_once('yocto_temperature.php');
cpp	#include "yocto_temperature.h"
m	#import "yocto_temperature.h"
pas	uses yocto_temperature;
vb	yocto_temperature.vb
cs	yocto_temperature.cs
java	import com.yoctopuce.YoctoAPI.YTemperature;
py	from yocto_temperature import *

Global functions

yFindTemperature(func)

Retrieves a temperature sensor for a given identifier.

yFirstTemperature()

Starts the enumeration of temperature sensors currently accessible.

`YTemperature` methods

temperature→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

temperature→describe()

Returns a short text that describes unambiguously the instance of the temperature sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

temperature→get_advertisedValue()

Returns the current value of the temperature sensor (no more than 6 characters).

temperature→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

temperature→get_currentValue()

Returns the current value of the temperature.

temperature→get_errorMessage()

Returns the error message of the latest error with the temperature sensor.

temperature→get_errorType()

Returns the numerical error code of the latest error with the temperature sensor.

temperature→get_friendlyName()

Returns a global identifier of the temperature sensor in the format `MODULE_NAME.FUNCTION_NAME`.

temperature→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

temperature→get_functionId()

Returns the hardware identifier of the temperature sensor, without reference to the module.

temperature→get_hardwareId()

Returns the unique hardware identifier of the temperature sensor in the form `SERIAL.FUNCTIONID`.

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()yFindTemperature() YTemperature

Retrieves a temperature sensor for a given identifier.

```
function yFindTemperature( 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() yFirstTemperature()yFirstTemperature() YTemperature

Starts the enumeration of temperature sensors currently accessible.

```
function yFirstTemperature( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues	array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
refValues	array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

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`.

```
function 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 temperature sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

temperature→get_advertisedValue() temperature→advertisedValue() temperature.get_advertisedValue() YTemperature

Returns the current value of the temperature sensor (no more than 6 characters).

```
function get_advertisedValue( )
```

Returns :

a string corresponding to the current value of the temperature sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

temperature→get_currentRawValue() temperature→currentRawValue() temperature.get_currentRawValue() YTemperature

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( )
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

temperature→get_currentValue() temperature→currentValue() temperature.get_currentValue() YTemperature

Returns the current value of the temperature.

```
function get_currentValue( )
```

Returns :

a floating point number corresponding to the current value of the temperature

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

temperature→get_errorMessage()
temperature→errorMessage()
temperature.get_errorMessage() YTemperature

Returns the error message of the latest error with the temperature sensor.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using 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.

```
function 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 occured while using the temperature sensor object

temperature→get_friendlyName() temperature→friendlyName() temperature.get_friendlyName()

YTemperature

Returns a global identifier of the temperature sensor in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the temperature sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the temperature sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the temperature sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

temperature→get_functionDescriptor() temperature→functionDescriptor() temperature.get_functionDescriptor() YTemperature

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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.

```
function get_functionId( )
```

For example `relay1`

Returns :

a string that identifies the temperature sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

temperature→get_hardwareId() temperature→hardwareId() temperature.get_hardwareId() YTemperature

Returns the unique hardware identifier of the temperature sensor in the form `SERIAL.FUNCTIONID`.

```
function 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() temperature→highestValue() temperature.get_highestValue() YTemperature

Returns the maximal value observed for the temperature since the device was started.

```
function 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() temperature→logFrequency() temperature.get_logFrequency() YTemperature

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function get_logFrequency( )
```

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

temperature→get_logicalName() temperature→logicalName() temperature.get_logicalName() YTemperature

Returns the logical name of the temperature sensor.

```
function 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.

```
function 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() temperature→module()temperature.get_module() YTemperature

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() temperature→module_async() temperature.get_module_async() YTemperature

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

temperature→get_recordedData() temperature→recordedData() temperature.get_recordedData()

YTemperature

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function 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.

temperature→get_reportFrequency() temperature→reportFrequency() temperature.get_reportFrequency() YTemperature

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function 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.

```
function get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

temperature→get_sensorType() temperature→sensorType() temperature.get_sensorType() YTemperature

Returns the temperature sensor type.

```
function 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()
temperature→unit()temperature.get_unit()

YTemperature

Returns the measuring unit for the temperature.

```
function 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`.

function **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.

```
function 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→isOnline_async() temperature.isOnline_async() YTemperature

Checks if the temperature sensor is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

temperature→load()temperature.load() YTemperature

Preloads the temperature sensor cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async()temperature.load_async() YTemperature

Preloads the temperature sensor cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

temperature→nextTemperature() temperature.nextTemperature()　　YTemperature

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

```
function 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.

```
function 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.

```
function 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.

```
function set_highestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_logFrequency() temperature→setLogFrequency() temperature.set_logFrequency() YTemperature

Changes the datalogger recording frequency for this function.

```
function set_logFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_logicalName() temperature→setLogicalName() temperature.set_logicalName() YTemperature

Changes the logical name of the temperature sensor.

```
function set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the temperature sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

temperature→set_lowestValue() temperature→setLowestValue() temperature.set_lowestValue() YTemperature

Changes the recorded minimal value observed.

```
function set_lowestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_reportFrequency() temperature→setReportFrequency() temperature.set_reportFrequency() YTemperature

Changes the timed value notification frequency for this function.

```
function 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.

```
function 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.

```
function 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() temperature→setUserData() temperature.set_userData()

YTemperature

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

temperature→wait_async()temperature.wait_async() YTemperature

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.39. Tilt function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	`<script type='text/javascript' src='yocto_tilt.js'></script>`

nodejs	var yoctolib = require('yoctolib');
	var YTilt = yoctolib.YTilt;

php	require_once('yocto_tilt.php');
cpp	#include "yocto_tilt.h"
m	#import "yocto_tilt.h"
pas	uses yocto_tilt;
vb	yocto_tilt.vb
cs	yocto_tilt.cs
java	import com.yoctopuce.YoctoAPI.YTilt;
py	from yocto_tilt import *

Global functions

yFindTilt(func)
Retrieves a tilt sensor for a given identifier.

yFirstTilt()
Starts the enumeration of tilt sensors currently accessible.

YTilt methods

tilt→calibrateFromPoints(rawValues, refValues)
Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

tilt→describe()
Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

tilt→get_advertisedValue()
Returns the current value of the tilt sensor (no more than 6 characters).

tilt→get_currentRawValue()
Returns the uncalibrated, unrounded raw value returned by the sensor.

tilt→get_currentValue()
Returns the current value of the inclination.

tilt→get_errorMessage()
Returns the error message of the latest error with the tilt sensor.

tilt→get_errorType()
Returns the numerical error code of the latest error with the tilt sensor.

tilt→get_friendlyName()
Returns a global identifier of the tilt sensor in the format `MODULE_NAME.FUNCTION_NAME`.

tilt→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

tilt→get_functionId()
Returns the hardware identifier of the tilt sensor, without reference to the module.

tilt→get_hardwareId()
Returns the unique hardware identifier of the tilt sensor in the form `SERIAL.FUNCTIONID`.

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.

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() yFindTilt()yFindTilt()

YTilt

Retrieves a tilt sensor for a given identifier.

```
function yFindTilt( 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() yFirstTilt()yFirstTilt()

YTilt

Starts the enumeration of tilt sensors currently accessible.

```
function yFirstTilt( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues	array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
refValues	array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→describe()tilt.describe() YTilt

Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function 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() tilt→advertisedValue()tilt.get_advertisedValue() YTilt

Returns the current value of the tilt sensor (no more than 6 characters).

function **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() tilt→currentRawValue()tilt.get_currentRawValue() YTilt

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function get_currentRawValue( )
```

Returns :
a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

tilt→get_currentValue() tilt→currentValue()tilt.get_currentValue() YTilt

Returns the current value of the inclination.

```
function get_currentValue( )
```

Returns :

a floating point number corresponding to the current value of the inclination

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

tilt→get_errorMessage() tilt→errorMessage()tilt.get_errorMessage() YTilt

Returns the error message of the latest error with the tilt sensor.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the tilt sensor object

tilt→get_errorType() tilt→errorType()tilt.get_errorType() YTilt

Returns the numerical error code of the latest error with the tilt sensor.

```
function 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 occured while using the tilt sensor object

tilt→get_friendlyName() tilt→friendlyName()tilt.get_friendlyName() YTilt

Returns a global identifier of the tilt sensor in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the tilt sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the tilt sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the tilt sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

tilt→get_functionDescriptor() tilt→functionDescriptor()tilt.get_functionDescriptor() YTilt

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() tilt→functionId()tilt.get_functionId() YTilt

Returns the hardware identifier of the tilt sensor, without reference to the module.

```
function 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() tilt→hardwareId()tilt.get_hardwareId() YTilt

Returns the unique hardware identifier of the tilt sensor in the form `SERIAL.FUNCTIONID`.

```
function 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() tilt→highestValue()tilt.get_highestValue() YTilt

Returns the maximal value observed for the inclination since the device was started.

```
function 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() tilt→logFrequency()tilt.get_logFrequency() YTilt

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function 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() tilt→logicalName()tilt.get_logicalName() YTilt

Returns the logical name of the tilt sensor.

```
function 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() tilt→lowestValue()tilt.get_lowestValue() YTilt

Returns the minimal value observed for the inclination since the device was started.

function **get_lowestValue**( )

Returns :

a floating point number corresponding to the minimal value observed for the inclination since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

tilt→get_module() tilt→module()tilt.get_module() YTilt

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() tilt→module_async()tilt.get_module_async() YTilt

Gets the YModule object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

tilt→get_recordedData() tilt→recordedData()tilt.get_recordedData() YTilt

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function 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.

tilt→get_reportFrequency() tilt→reportFrequency()tilt.get_reportFrequency() YTilt

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function 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() tilt→resolution()tilt.get_resolution() **YTilt**

Returns the resolution of the measured values.

```
function get_resolution( )
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

tilt→get_unit() tilt→unit()tilt.get_unit() YTilt

Returns the measuring unit for the inclination.

```
function 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() tilt→userData()tilt.get_userData() YTilt

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **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.

```
function 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→isOnline_async()tilt.isOnline_async() YTilt

Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

tilt→load()tilt.load() YTilt

Preloads the tilt sensor cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async()tilt.load_async() YTilt

Preloads the tilt sensor cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

tilt→nextTilt()tilt.nextTilt() YTilt

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

```
function 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.

```
function 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.

```
function 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()
tilt→setHighestValue()tilt.set_highestValue() YTilt

Changes the recorded maximal value observed.

```
function 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() tilt→setLogFrequency()tilt.set_logFrequency() YTilt

Changes the datalogger recording frequency for this function.

```
function 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() tilt→setLogicalName()tilt.set_logicalName() YTilt

Changes the logical name of the tilt sensor.

```
function 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() tilt→setLowestValue()tilt.set_lowestValue() YTilt

Changes the recorded minimal value observed.

```
function 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() tilt→setReportFrequency()tilt.set_reportFrequency() YTilt

Changes the timed value notification frequency for this function.

```
function 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() tilt→setResolution()tilt.set_resolution() YTilt

Changes the resolution of the measured physical values.

```
function 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() tilt→setUserData()tilt.set_userData() YTilt

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

tilt→wait_async()tilt.wait_async() YTilt

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.40. Voc function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_voc.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YVoc = yoctolib.YVoc;

php	require_once('yocto_voc.php');
cpp	#include "yocto_voc.h"
m	#import "yocto_voc.h"
pas	uses yocto_voc;
vb	yocto_voc.vb
cs	yocto_voc.cs
java	import com.yoctopuce.YoctoAPI.YVoc;
py	from yocto_voc import *

Global functions
yFindVoc(func) Retrieves a Volatile Organic Compound sensor for a given identifier.
yFirstVoc() Starts the enumeration of Volatile Organic Compound sensors currently accessible.
YVoc methods
voc→calibrateFromPoints(rawValues, refValues) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
voc→describe() Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
voc→get_advertisedValue() Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).
voc→get_currentRawValue() Returns the unrounded and uncalibrated raw value returned by the sensor.
voc→get_currentValue() Returns the current measure for the estimated VOC concentration.
voc→get_errorMessage() Returns the error message of the latest error with the Volatile Organic Compound sensor.
voc→get_errorType() Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.
voc→get_friendlyName() Returns a global identifier of the Volatile Organic Compound sensor in the format `MODULE_NAME.FUNCTION_NAME`.
voc→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
voc→get_functionId() Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.
voc→get_hardwareId()

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form `SERIAL.FUNCTIONID`.

voc→get_highestValue()

Returns the maximal value observed for the estimated VOC concentration.

voc→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voc→get_logicalName()

Returns the logical name of the Volatile Organic Compound sensor.

voc→get_lowestValue()

Returns the minimal value observed for the estimated VOC concentration.

voc→get_module()

Gets the `YModule` object for the device on which the function is located.

voc→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

voc→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

voc→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voc→get_resolution()

Returns the resolution of the measured values.

voc→get_unit()

Returns the measuring unit for the estimated VOC concentration.

voc→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

voc→isOnline()

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

voc→isOnline_async(callback, context)

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error (asynchronous version).

voc→load(msValidity)

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

voc→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

voc→load_async(msValidity, callback, context)

Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).

voc→nextVoc()

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

voc→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

voc→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

voc→set_highestValue(newval)

Changes the recorded maximal value observed for the estimated VOC concentration.

voc→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

voc→set_logicalName(newval)

Changes the logical name of the Volatile Organic Compound sensor.

voc→set_lowestValue(newval)

Changes the recorded minimal value observed for the estimated VOC concentration.

voc→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voc→set_resolution(newval)

Changes the resolution of the measured physical values.

voc→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voc→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoc.FindVoc() yFindVoc()yFindVoc()

YVoc

Retrieves a Volatile Organic Compound sensor for a given identifier.

```
function yFindVoc( 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()yFirstVoc() YVoc

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

```
function yFirstVoc( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues	array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
refValues	array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

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`.

```
function 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 Volatile Organic Compound sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

voc→get_advertisedValue() voc→advertisedValue()voc.get_advertisedValue() YVoc

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

```
function 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() voc→currentRawValue()voc.get_currentRawValue() YVoc

Returns the unrounded and uncalibrated raw value returned by the sensor.

```
function get_currentRawValue( )
```

Returns :
a floating point number corresponding to the unrounded and uncalibrated raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

voc→get_currentValue() voc→currentValue()voc.get_currentValue() YVoc

Returns the current measure for the estimated VOC concentration.

function **get_currentValue**( )

Returns :

a floating point number corresponding to the current measure for the estimated VOC concentration

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

voc→get_errorMessage() voc→errorMessage()voc.get_errorMessage() YVoc

Returns the error message of the latest error with the Volatile Organic Compound sensor.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occured while using the Volatile Organic Compound sensor object

voc→get_errorType() voc→errorType()voc.get_errorType() YVoc

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

function **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 occured while using the Volatile Organic Compound sensor object

voc→get_friendlyName() voc→friendlyName()voc.get_friendlyName() YVoc

Returns a global identifier of the Volatile Organic Compound sensor in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the Volatile Organic Compound sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the Volatile Organic Compound sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the Volatile Organic Compound sensor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

voc→get_functionDescriptor() voc→functionDescriptor() voc.get_functionDescriptor() YVoc

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() voc→functionId()voc.get_functionId() YVoc

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

```
function 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() voc→hardwareId()voc.get_hardwareId() YVoc

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form `SERIAL.FUNCTIONID`.

```
function 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() voc→highestValue()voc.get_highestValue() YVoc

Returns the maximal value observed for the estimated VOC concentration.

```
function get_highestValue( )
```

Returns :

a floating point number corresponding to the maximal value observed for the estimated VOC concentration

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

voc→get_logFrequency() voc→logFrequency()voc.get_logFrequency() YVoc

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function 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() voc→logicalName()voc.get_logicalName() YVoc

Returns the logical name of the Volatile Organic Compound sensor.

function **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() voc→lowestValue()voc.get_lowestValue() YVoc

Returns the minimal value observed for the estimated VOC concentration.

```
function get_lowestValue( )
```

Returns :

a floating point number corresponding to the minimal value observed for the estimated VOC concentration

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

voc→get_module() voc→module()voc.get_module() YVoc

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() voc→module_async()voc.get_module_async() YVoc

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

voc→get_recordedData() voc→recordedData()voc.get_recordedData() YVoc

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function get_recordedData( startTime, endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime	the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.
endTime	the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

voc→get_reportFrequency() voc→reportFrequency()voc.get_reportFrequency() YVoc

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function get_reportFrequency( )
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

voc→get_resolution() voc→resolution()voc.get_resolution() YVoc

Returns the resolution of the measured values.

```
function 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() voc→unit()voc.get_unit() YVoc

Returns the measuring unit for the estimated VOC concentration.

```
function get_unit( )
```

Returns :
a string corresponding to the measuring unit for the estimated VOC concentration

On failure, throws an exception or returns `Y_UNIT_INVALID`.

voc→get_userData() voc→userData()voc.get_userData() YVoc

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function 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.

```
function 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→isOnline_async()voc.isOnline_async() YVoc

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

voc→load()voc.load() YVoc

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

```
function 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 trafic 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`.

```
function 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→load_async()voc.load_async() YVoc

Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

voc→nextVoc()voc.nextVoc() YVoc

Continues the enumeration of Volatile Organic Compound sensors started using `yFirstVoc()`.

```
function 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.

```
function 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.

```
function 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() voc→setHighestValue()voc.set_highestValue() YVoc

Changes the recorded maximal value observed for the estimated VOC concentration.

```
function set_highestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed for the estimated VOC concentration

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_logFrequency() voc→setLogFrequency()voc.set_logFrequency() YVoc

Changes the datalogger recording frequency for this function.

```
function 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() voc→setLogicalName()voc.set_logicalName() YVoc

Changes the logical name of the Volatile Organic Compound sensor.

```
function 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() voc→setLowestValue()voc.set_lowestValue() YVoc

Changes the recorded minimal value observed for the estimated VOC concentration.

```
function set_lowestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed for the estimated VOC concentration

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_reportFrequency() voc→setReportFrequency() voc.set_reportFrequency() YVoc

Changes the timed value notification frequency for this function.

```
function set_reportFrequency( newval)
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_resolution() voc→setResolution()voc.set_resolution() YVoc

Changes the resolution of the measured physical values.

```
function 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() voc→setUserData()voc.set_userData() YVoc

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

voc→wait_async()voc.wait_async() YVoc

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.41. Voltage function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_voltage.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YVoltage = yoctolib.YVoltage;

php	require_once('yocto_voltage.php');
cpp	#include "yocto_voltage.h"
m	#import "yocto_voltage.h"
pas	uses yocto_voltage;
vb	yocto_voltage.vb
cs	yocto_voltage.cs
java	import com.yoctopuce.YoctoAPI.YVoltage;
py	from yocto_voltage import *

Global functions

yFindVoltage(func)

Retrieves a voltage sensor for a given identifier.

yFirstVoltage()

Starts the enumeration of voltage sensors currently accessible.

`YVoltage` methods

voltage→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

voltage→describe()

Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

voltage→get_advertisedValue()

Returns the current value of the voltage sensor (no more than 6 characters).

voltage→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor.

voltage→get_currentValue()

Returns the current measure for the voltage.

voltage→get_errorMessage()

Returns the error message of the latest error with the voltage sensor.

voltage→get_errorType()

Returns the numerical error code of the latest error with the voltage sensor.

voltage→get_friendlyName()

Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.

voltage→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

voltage→get_functionId()

Returns the hardware identifier of the voltage sensor, without reference to the module.

voltage→get_hardwareId()

Returns the unique hardware identifier of the voltage sensor in the form `SERIAL.FUNCTIONID`.

voltage→get_highestValue()

Returns the maximal value observed for the voltage.

voltage→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voltage→get_logicalName()

Returns the logical name of the voltage sensor.

voltage→get_lowestValue()

Returns the minimal value observed for the voltage.

voltage→get_module()

Gets the `YModule` object for the device on which the function is located.

voltage→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

voltage→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

voltage→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voltage→get_resolution()

Returns the resolution of the measured values.

voltage→get_unit()

Returns the measuring unit for the voltage.

voltage→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

voltage→isOnline()

Checks if the voltage sensor is currently reachable, without raising any error.

voltage→isOnline_async(callback, context)

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

voltage→load(msValidity)

Preloads the voltage sensor cache with a specified validity duration.

voltage→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

voltage→load_async(msValidity, callback, context)

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

voltage→nextVoltage()

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

voltage→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

voltage→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

voltage→set_highestValue(newval)

Changes the recorded maximal value observed pour the voltage.

voltage→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

voltage→set_logicalName(newval)

Changes the logical name of the voltage sensor.

voltage→set_lowestValue(newval)

Changes the recorded minimal value observed pour the voltage.

voltage→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voltage→set_resolution(newval)

Changes the resolution of the measured values.

voltage→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voltage→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoltage.FindVoltage() yFindVoltage()yFindVoltage() YVoltage

Retrieves a voltage sensor for a given identifier.

```
function yFindVoltage( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltage.isOnline()` to test if the voltage sensor is indeed online at a given time. In case of ambiguity when looking for a voltage sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the voltage sensor

Returns :

a `YVoltage` object allowing you to drive the voltage sensor.

YVoltage.FirstVoltage() yFirstVoltage()yFirstVoltage() YVoltage

Starts the enumeration of voltage sensors currently accessible.

```
function yFirstVoltage( )
```

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.

```
function calibrateFromPoints( rawValues, refValues)
```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues	array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.
refValues	array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

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`.

```
function 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()
voltage→advertisedValue()
voltage.get_advertisedValue()

YVoltage

Returns the current value of the voltage sensor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the voltage sensor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

voltage→get_currentRawValue() voltage→currentRawValue() voltage.get_currentRawValue() YVoltage

Returns the uncalibrated, unrounded raw value returned by the sensor.

```
function 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`.

voltage→get_currentValue() voltage→currentValue()voltage.get_currentValue() YVoltage

Returns the current measure for the voltage.

```
function get_currentValue( )
```

Returns :
a floating point number corresponding to the current measure for the voltage

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

voltage→get_errorMessage() voltage→errorMessage()voltage.get_errorMessage() YVoltage

Returns the error message of the latest error with the voltage sensor.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the voltage sensor object

voltage→get_errorType() voltage→errorType()voltage.get_errorType() YVoltage

Returns the numerical error code of the latest error with the voltage sensor.

```
function 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 occured while using the voltage sensor object

voltage→get_friendlyName() voltage→friendlyName()voltage.get_friendlyName() **YVoltage**

Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the voltage sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage sensor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the voltage sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

voltage→get_functionDescriptor() voltage→functionDescriptor() voltage.get_functionDescriptor() YVoltage

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() voltage→functionId()voltage.get_functionId() YVoltage

Returns the hardware identifier of the voltage sensor, without reference to the module.

```
function 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() voltage→hardwareId()voltage.get_hardwareId() YVoltage

Returns the unique hardware identifier of the voltage sensor in the form `SERIAL.FUNCTIONID`.

```
function 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() voltage→highestValue()voltage.get_highestValue()

YVoltage

Returns the maximal value observed for the voltage.

```
function get_highestValue( )
```

Returns :

a floating point number corresponding to the maximal value observed for the voltage

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

voltage→get_logFrequency() voltage→logFrequency()voltage.get_logFrequency() YVoltage

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
function 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() voltage→logicalName()voltage.get_logicalName() YVoltage

Returns the logical name of the voltage sensor.

```
function 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() voltage→lowestValue()voltage.get_lowestValue() YVoltage

Returns the minimal value observed for the voltage.

```
function get_lowestValue( )
```

Returns :
a floating point number corresponding to the minimal value observed for the voltage

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

voltage→get_module() voltage→module()voltage.get_module() YVoltage

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() voltage→module_async() voltage.get_module_async() YVoltage

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

voltage→get_recordedData() voltage→recordedData()voltage.get_recordedData() YVoltage

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
function 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.

voltage→get_reportFrequency()
voltage→reportFrequency()
voltage.get_reportFrequency()

YVoltage

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
function 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() voltage→resolution()voltage.get_resolution() YVoltage

Returns the resolution of the measured values.

```
function 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() voltage→unit()voltage.get_unit() YVoltage

Returns the measuring unit for the voltage.

```
function 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() voltage→userData()voltage.get_userData() YVoltage

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function 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.

```
function 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→isOnline_async()voltage.isOnline_async() YVoltage

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

voltage→load()voltage.load() YVoltage

Preloads the voltage sensor cache with a specified validity duration.

```
function 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 trafic 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() voltage.loadCalibrationPoints() YVoltage

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
function 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→load_async()voltage.load_async() YVoltage

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

function **load_async**(**msValidity**, **callback**, **context**)

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

voltage→nextVoltage()voltage.nextVoltage() YVoltage

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

```
function 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.

```
function 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.

```
function registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

voltage→set_highestValue()
voltage→setHighestValue()
voltage.set_highestValue() YVoltage

Changes the recorded maximal value observed pour the voltage.

```
function set_highestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed pour the voltage

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_logFrequency()
voltage→setLogFrequency()
voltage.set_logFrequency() YVoltage

Changes the datalogger recording frequency for this function.

```
function 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() voltage→setLogicalName()voltage.set_logicalName() YVoltage

Changes the logical name of the voltage sensor.

function **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() voltage→setLowestValue()voltage.set_lowestValue() YVoltage

Changes the recorded minimal value observed pour the voltage.

```
function set_lowestValue( newval)
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed pour the voltage

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_reportFrequency() voltage→setReportFrequency() voltage.set_reportFrequency() YVoltage

Changes the timed value notification frequency for this function.

```
function 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() voltage→setResolution()voltage.set_resolution() YVoltage

Changes the resolution of the measured values.

```
function set_resolution( newval)
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_userData() voltage→setUserData()voltage.set_userData() YVoltage

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

voltage→wait_async()voltage.wait_async() YVoltage

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.42. Voltage source function interface

Yoctopuce application programming interface allows you to control the module voltage output. You affect absolute output values or make transitions

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_vsource.js'></script>
php	require_once('yocto_vsource.php');
cpp	#include "yocto_vsource.h"
m	#import "yocto_vsource.h"
pas	uses yocto_vsource;
vb	yocto_vsource.vb
cs	yocto_vsource.cs
java	import com.yoctopuce.YoctoAPI.YVSource;
py	from yocto_vsource import *

Global functions

yFindVSource(func)

Retrieves a voltage source for a given identifier.

yFirstVSource()

Starts the enumeration of voltage sources currently accessible.

`YVSource` methods

vsource→describe()

Returns a short text that describes the function in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

vsource→get_advertisedValue()

Returns the current value of the voltage source (no more than 6 characters).

vsource→get_errorMessage()

Returns the error message of the latest error with this function.

vsource→get_errorType()

Returns the numerical error code of the latest error with this function.

vsource→get_extPowerFailure()

Returns true if external power supply voltage is too low.

vsource→get_failure()

Returns true if the module is in failure mode.

vsource→get_friendlyName()

Returns a global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

vsource→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

vsource→get_functionId()

Returns the hardware identifier of the function, without reference to the module.

vsource→get_hardwareId()

Returns the unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

vsource→get_logicalName()

Returns the logical name of the voltage source.

vsource→get_module()

Gets the `YModule` object for the device on which the function is located.

vsource→get_module_async(callback, context)

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 maintaint 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.FindVSource()yFindVSource() YVSource

Retrieves a voltage source for a given identifier.

function **yFindVSource**(**func**)

yFindVSource() — YVSource.FindVSource()yFindVSource()

Retrieves a voltage source for a given identifier.

js	function **yFindVSource**(**func**)
php	function **yFindVSource**(**$func**)
cpp	YVSource* **yFindVSource**(const string& **func**)
m	YVSource* **yFindVSource**(NSString* **func**)
pas	function **yFindVSource**(**func**: string): TYVSource
vb	function **yFindVSource**(ByVal **func** As String) As YVSource
cs	YVSource **FindVSource**(string **func**)
java	YVSource **FindVSource**(String **func**)
py	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.FirstVSource()yFirstVSource() YVSource

Starts the enumeration of voltage sources currently accessible.

function **yFirstVSource**( )

yFirstVSource() — YVSource.FirstVSource()yFirstVSource()

Starts the enumeration of voltage sources currently accessible.

`js`	function **yFirstVSource**( )
`php`	function **yFirstVSource**( )
`cpp`	YVSource* **yFirstVSource**( )
`m`	YVSource* **yFirstVSource**( )
`pas`	function **yFirstVSource**( ): TYVSource
`vb`	function **yFirstVSource**( ) As YVSource
`cs`	YVSource **FirstVSource**( )
`java`	YVSource **FirstVSource**( )
`py`	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→describe()vsource.describe() YVSource

Returns a short text that describes the function in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

function **describe**( )

vsource→describe()vsource.describe()

Returns a short text that describes the function in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

js	function **describe**( )
php	function **describe**( )
cpp	string **describe**( )
m	-(NSString*) **describe**
pas	function **describe**( ): string
vb	function **describe**( ) As String
cs	string **describe**( )
java	String **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 function (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

vsource→get_advertisedValue() vsource→advertisedValue() vsource.get_advertisedValue() YVSource

Returns the current value of the voltage source (no more than 6 characters).

function **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 **get_advertisedValue**( )
php	function **get_advertisedValue**( )
cpp	string **get_advertisedValue**( )
m	-(NSString*) advertisedValue
pas	function **get_advertisedValue**( ): string
vb	function **get_advertisedValue**( ) As String
cs	string **get_advertisedValue**( )
java	String **get_advertisedValue**( )
py	def **get_advertisedValue**( )
cmd	YVSource **target** **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`.

vsource→get_errorMessage() vsource→errorMessage() vsource.get_errorMessage() — YVSource

Returns the error message of the latest error with this function.

function **get_errorMessage**( )

vsource→get_errorMessage() vsource→errorMessage()vsource.get_errorMessage()

Returns the error message of the latest error with this function.

`js`	function **get_errorMessage**( )
`php`	function **get_errorMessage**( )
`cpp`	string **get_errorMessage**( )
`m`	-(NSString*) errorMessage
`pas`	function **get_errorMessage**( ): string
`vb`	function **get_errorMessage**( ) As String
`cs`	string **get_errorMessage**( )
`java`	String **get_errorMessage**( )
`py`	def **get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using this function object

vsource→get_errorType() vsource→errorType()vsource.get_errorType() YVSource

Returns the numerical error code of the latest error with this function.

function **get_errorType**( )

vsource→get_errorType() vsource→errorType()vsource.get_errorType()

Returns the numerical error code of the latest error with this function.

js	function **get_errorType**( )
php	function **get_errorType**( )
cpp	YRETCODE **get_errorType**( )
pas	function **get_errorType**( ): YRETCODE
vb	function **get_errorType**( ) As YRETCODE
cs	YRETCODE **get_errorType**( )
java	int **get_errorType**( )
py	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 occured while using this function object

vsource→get_extPowerFailure()
vsource→extPowerFailure()
vsource.get_extPowerFailure() YVSource

Returns true if external power supply voltage is too low.

function **get_extPowerFailure**( )

vsource→get_extPowerFailure()
vsource→extPowerFailure()vsource.get_extPowerFailure()

Returns true if external power supply voltage is too low.

js	function **get_extPowerFailure**( )
php	function **get_extPowerFailure**( )
cpp	Y_EXTPOWERFAILURE_enum **get_extPowerFailure**( )
m	-(Y_EXTPOWERFAILURE_enum) extPowerFailure
pas	function **get_extPowerFailure**( ): Integer
vb	function **get_extPowerFailure**( ) As Integer
cs	int **get_extPowerFailure**( )
java	int **get_extPowerFailure**( )
py	def **get_extPowerFailure**( )
cmd	YVSource **target** **get_extPowerFailure**

Returns :

either `Y_EXTPOWERFAILURE_FALSE` or `Y_EXTPOWERFAILURE_TRUE`, according to true if external power supply voltage is too low

On failure, throws an exception or returns `Y_EXTPOWERFAILURE_INVALID`.

vsource→get_failure() vsource→failure()vsource.get_failure() YVSource

Returns true if the module is in failure mode.

function **get_failure**( )

vsource→get_failure() vsource→failure()vsource.get_failure()

Returns true if the module is in failure mode.

js	function **get_failure**( )
php	function **get_failure**( )
cpp	Y_FAILURE_enum **get_failure**( )
m	-(Y_FAILURE_enum) failure
pas	function **get_failure**( ): Integer
vb	function **get_failure**( ) As Integer
cs	int **get_failure**( )
java	int **get_failure**( )
py	def **get_failure**( )
cmd	YVSource **target** **get_failure**

More information can be obtained by testing get_overheat, get_overcurrent etc... When a error condition is met, the output voltage is set to 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_friendlyName() vsource→friendlyName()vsource.get_friendlyName() YVSource

Returns a global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

vsource→get_friendlyName() vsource→friendlyName()vsource.get_friendlyName()

Returns a global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

`js`	function **get_friendlyName**( )
`php`	function **get_friendlyName**( )
`cpp`	virtual string **get_friendlyName**( )
`m`	-(NSString*) friendlyName
`cs`	override string **get_friendlyName**( )
`java`	String **get_friendlyName**( )

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the function using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

vsource→get_functionDescriptor() vsource→functionDescriptor() vsource.get_vsourceDescriptor() YVSource

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

function **get_functionDescriptor**( )

vsource→get_functionDescriptor() vsource→functionDescriptor()vsource.get_vsourceDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function **get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	function **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	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_functionId() vsource→functionId()vsource.get_vsourceId() YVSource

Returns the hardware identifier of the function, without reference to the module.

function **get_functionId**( )

vsource→get_functionId() vsource→functionId()vsource.get_vsourceId()

Returns the hardware identifier of the function, without reference to the module.

`js` function **get_functionId**( )
`php` function **get_functionId**( )
`cpp` string **get_functionId**( )
`m` -(NSString*) functionId
`vb` function **get_functionId**( ) As String
`cs` string **get_functionId**( )
`java` String **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the function (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

vsource→get_hardwareId() vsource→hardwareId()vsource.get_hardwareId() YVSource

Returns the unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

function **get_hardwareId**( )

vsource→get_hardwareId() vsource→hardwareId()vsource.get_hardwareId()

Returns the unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

js	function **get_hardwareId**( )
php	function **get_hardwareId**( )
cpp	string **get_hardwareId**( )
m	-(NSString*) hardwareId
vb	function **get_hardwareId**( ) As String
cs	string **get_hardwareId**( )
java	String **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the function (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

vsource→get_logicalName() vsource→logicalName()vsource.get_logicalName() YVSource

Returns the logical name of the voltage source.

function **get_logicalName**( )

vsource→get_logicalName() vsource→logicalName()vsource.get_logicalName()

Returns the logical name of the voltage source.

js	function **get_logicalName**( )
php	function **get_logicalName**( )
cpp	string **get_logicalName**( )
m	-(NSString*) logicalName
pas	function **get_logicalName**( ): string
vb	function **get_logicalName**( ) As String
cs	string **get_logicalName**( )
java	String **get_logicalName**( )
py	def **get_logicalName**( )
cmd	YVSource **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the voltage source

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

vsource→get_module() vsource→module()vsource.get_module() YVSource

Gets the `YModule` object for the device on which the function is located.

function **get_module**( )

vsource→get_module() vsource→module()vsource.get_module()

Gets the `YModule` object for the device on which the function is located.

js	function **get_module**( )
php	function **get_module**( )
cpp	YModule * **get_module**( )
m	-(YModule*) module
pas	function **get_module**( ): TYModule
vb	function **get_module**( ) As YModule
cs	YModule **get_module**( )
java	YModule **get_module**( )
py	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_module_async() vsource→module_async() vsource.get_module_async() YVSource

Gets the `YModule` object for the device on which the function is located (asynchronous version).

function **get_module_async**(**callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

vsource→get_overCurrent() vsource→overCurrent()vsource.get_overCurrent() YVSource

Returns true if the appliance connected to the device is too greedy .

function **get_overCurrent**( )

vsource→get_overCurrent() vsource→overCurrent()vsource.get_overCurrent()

Returns true if the appliance connected to the device is too greedy .

`js`	function **get_overCurrent**( )
`php`	function **get_overCurrent**( )
`cpp`	Y_OVERCURRENT_enum **get_overCurrent**( )
`m`	-(Y_OVERCURRENT_enum) overCurrent
`pas`	function **get_overCurrent**( ): Integer
`vb`	function **get_overCurrent**( ) As Integer
`cs`	int **get_overCurrent**( )
`java`	int **get_overCurrent**( )
`py`	def **get_overCurrent**( )
`cmd`	YVSource **target** **get_overCurrent**

Returns :

either `Y_OVERCURRENT_FALSE` or `Y_OVERCURRENT_TRUE`, according to true if the appliance connected to the device is too greedy

On failure, throws an exception or returns `Y_OVERCURRENT_INVALID`.

vsource→get_overHeat() vsource→overHeat()vsource.get_overHeat() YVSource

Returns TRUE if the module is overheating.

function **get_overHeat**( )

vsource→get_overHeat() vsource→overHeat()vsource.get_overHeat()

Returns TRUE if the module is overheating.

`js`	function **get_overHeat**( )
`php`	function **get_overHeat**( )
`cpp`	Y_OVERHEAT_enum **get_overHeat**( )
`m`	-(Y_OVERHEAT_enum) overHeat
`pas`	function **get_overHeat**( ): Integer
`vb`	function **get_overHeat**( ) As Integer
`cs`	int **get_overHeat**( )
`java`	int **get_overHeat**( )
`py`	def **get_overHeat**( )
`cmd`	YVSource **target** **get_overHeat**

Returns :

either `Y_OVERHEAT_FALSE` or `Y_OVERHEAT_TRUE`, according to TRUE if the module is overheating

On failure, throws an exception or returns `Y_OVERHEAT_INVALID`.

vsource→get_overLoad() vsource→overLoad()vsource.get_overLoad() YVSource

Returns true if the device is not able to maintaint the requested voltage output .

function **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 **get_overLoad**( )
php	function **get_overLoad**( )
cpp	Y_OVERLOAD_enum **get_overLoad**( )
m	-(Y_OVERLOAD_enum) overLoad
pas	function **get_overLoad**( ): Integer
vb	function **get_overLoad**( ) As Integer
cs	int **get_overLoad**( )
java	int **get_overLoad**( )
py	def **get_overLoad**( )
cmd	YVSource **target** **get_overLoad**

Returns :

either `Y_OVERLOAD_FALSE` or `Y_OVERLOAD_TRUE`, according to true if the device is not able to maintaint the requested voltage output

On failure, throws an exception or returns `Y_OVERLOAD_INVALID`.

vsource→get_regulationFailure() vsource→regulationFailure() vsource.get_regulationFailure()

YVSource

Returns true if the voltage output is too high regarding the requested voltage .

```
function 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 **get_regulationFailure**( )
php	function **get_regulationFailure**( )
cpp	Y_REGULATIONFAILURE_enum **get_regulationFailure**( )
m	-(Y_REGULATIONFAILURE_enum) regulationFailure
pas	function **get_regulationFailure**( ): Integer
vb	function **get_regulationFailure**( ) As Integer
cs	int **get_regulationFailure**( )
java	int **get_regulationFailure**( )
py	def **get_regulationFailure**( )
cmd	YVSource **target** **get_regulationFailure**

Returns :

either `Y_REGULATIONFAILURE_FALSE` or `Y_REGULATIONFAILURE_TRUE`, according to true if the voltage output is too high regarding the requested voltage

On failure, throws an exception or returns `Y_REGULATIONFAILURE_INVALID`.

vsource→get_unit() vsource→unit()vsource.get_unit()

YVSource

Returns the measuring unit for the voltage.

function **get_unit**( )

vsource→get_unit() vsource→unit()vsource.get_unit()

Returns the measuring unit for the voltage.

js	function **get_unit**( )
php	function **get_unit**( )
cpp	string **get_unit**( )
m	-(NSString*) unit
pas	function **get_unit**( ): string
vb	function **get_unit**( ) As String
cs	string **get_unit**( )
java	String **get_unit**( )
py	def **get_unit**( )
cmd	YVSource **target** **get_unit**

Returns :
a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns `Y_UNIT_INVALID`.

vsource→get_userData() vsource→userData()vsource.get_userData()

YVSource

Returns the value of the userData attribute, as previously stored using method `set_userData`.

function **get_userData**( )

vsource→get_userData() vsource→userData()vsource.get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

`js`	function **get_userData**( )
`php`	function **get_userData**( )
`cpp`	void * **get_userData**( )
`m`	-(void*) userData
`pas`	function **get_userData**( ): Tobject
`vb`	function **get_userData**( ) As Object
`cs`	object **get_userData**( )
`java`	Object **get_userData**( )
`py`	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() vsource→voltage()vsource.get_voltage() YVSource

Returns the voltage output command (mV)

function **get_voltage**( )

vsource→get_voltage() vsource→voltage()vsource.get_voltage()

Returns the voltage output command (mV)

`js`	function **get_voltage**( )
`php`	function **get_voltage**( )
`cpp`	int **get_voltage**( )
`m`	-(int) voltage
`pas`	function **get_voltage**( ): LongInt
`vb`	function **get_voltage**( ) As Integer
`cs`	int **get_voltage**( )
`java`	int **get_voltage**( )
`py`	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.

function **isOnline**( )

vsource→isOnline()vsource.isOnline()

Checks if the function is currently reachable, without raising any error.

`js`	function **isOnline**( )
`php`	function **isOnline**( )
`cpp`	bool **isOnline**( )
`m`	-(BOOL) **isOnline**
`pas`	function **isOnline**( ): boolean
`vb`	function **isOnline**( ) As Boolean
`cs`	bool **isOnline**( )
`java`	boolean **isOnline**( )
`py`	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→isOnline_async()vsource.isOnline_async() YVSource

Checks if the function is currently reachable, without raising any error (asynchronous version).

function **isOnline_async**(**callback**, **context**)

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.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

vsource→load()vsource.load() YVSource

Preloads the function cache with a specified validity duration.

function **load**(**msValidity**)

vsource→load()vsource.load()

Preloads the function cache with a specified validity duration.

js	function **load**(**msValidity**)
php	function **load**(**$msValidity**)
cpp	YRETCODE **load**(int **msValidity**)
m	-(YRETCODE) **load** : (int) **msValidity**
pas	function **load**(**msValidity**: integer): YRETCODE
vb	function **load**(ByVal **msValidity** As Integer) As YRETCODE
cs	YRETCODE **load**(int **msValidity**)
java	int **load**(long **msValidity**)
py	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 trafic 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→load_async()vsource.load_async() YVSource

Preloads the function cache with a specified validity duration (asynchronous version).

function **load_async**(**msValidity**, **callback**, **context**)

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

vsource→nextVSource()vsource.nextVSource() YVSource

Continues the enumeration of voltage sources started using `yFirstVSource()`.

function **nextVSource**( )

vsource→nextVSource()vsource.nextVSource()

Continues the enumeration of voltage sources started using `yFirstVSource()`.

`js`	function **nextVSource**( )
`php`	function **nextVSource**( )
`cpp`	YVSource * **nextVSource**( )
`m`	-(YVSource*) **nextVSource**
`pas`	function **nextVSource**( ): TYVSource
`vb`	function **nextVSource**( ) As YVSource
`cs`	YVSource **nextVSource**( )
`java`	YVSource **nextVSource**( )
`py`	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.

function **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 **pulse**(**voltage**, **ms_duration**)
php	function **pulse**(**$voltage**, **$ms_duration**)
cpp	int **pulse**(int **voltage**, int **ms_duration**)
m	-(int) **pulse** : (int) **voltage** : (int) **ms_duration**
pas	function **pulse**(**voltage**: integer, **ms_duration**: integer): integer
vb	function **pulse**(ByVal **voltage** As Integer, ByVal **ms_duration** As Integer) As Integer
cs	int **pulse**(int **voltage**, int **ms_duration**)
java	int **pulse**(int **voltage**, int **ms_duration**)
py	def **pulse**(**voltage**, **ms_duration**)
cmd	YVSource **target** **pulse** **voltage** **ms_duration**

Parameters :

voltage	pulse voltage, in millivolts
ms_duration	pulse duration, in millisecondes

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.

function **registerValueCallback**(**callback**)

vsource→registerValueCallback()vsource.registerValueCallback()

Registers the callback function that is invoked on every change of advertised value.

`js`	function **registerValueCallback**(**callback**)
`php`	function **registerValueCallback**(**$callback**)
`cpp`	void **registerValueCallback**(YDisplayUpdateCallback **callback**)
`pas`	procedure **registerValueCallback**(**callback**: TGenericUpdateCallback)
`vb`	procedure **registerValueCallback**(ByVal **callback** As GenericUpdateCallback)
`cs`	void **registerValueCallback**(UpdateCallback **callback**)
`java`	void **registerValueCallback**(UpdateCallback **callback**)
`py`	def **registerValueCallback**(**callback**)
`m`	-(void) **registerValueCallback** : (YFunctionUpdateCallback) **callback**

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

vsource→set_logicalName() vsource→setLogicalName() vsource.set_logicalName() YVSource

Changes the logical name of the voltage source.

function **set_logicalName**(**newval**)

vsource→set_logicalName() vsource→setLogicalName()vsource.set_logicalName()

Changes the logical name of the voltage source.

js	function **set_logicalName**(**newval**)
php	function **set_logicalName**(**$newval**)
cpp	int **set_logicalName**(const string& **newval**)
m	-(int) setLogicalName : (NSString*) **newval**
pas	function **set_logicalName**(**newval**: string): integer
vb	function **set_logicalName**(ByVal **newval** As String) As Integer
cs	int **set_logicalName**(string **newval**)
java	int **set_logicalName**(String **newval**)
py	def **set_logicalName**(**newval**)
cmd	YVSource **target** **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.

vsource→set_userData() vsource→setUserData()vsource.set_userData() YVSource

Stores a user context provided as argument in the userData attribute of the function.

```
function set_userData( data)
```

vsource→set_userData() vsource→setUserData()vsource.set_userData()

Stores a user context provided as argument in the userData attribute of the function.

```
js    function set_userData( data)
php   function set_userData( $data)
cpp   void set_userData( void* data)
m     -(void) setUserData : (void*) data
pas   procedure set_userData( data: Tobject)
vb    procedure set_userData( ByVal data As Object)
cs    void set_userData( object data)
java  void set_userData( Object data)
py    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() vsource→setVoltage()vsource.set_voltage() YVSource

Tunes the device output voltage (milliVolts).

```
function 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.

```
function 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.

vsource→wait_async()vsource.wait_async() YVSource

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

function **wait_async**(**callback**, **context**)

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing :

3.43. WakeUpMonitor function interface

The WakeUpMonitor function handles globally all wake-up sources, as well as automated sleep mode.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wakeupmonitor.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YWakeUpMonitor = yoctolib.YWakeUpMonitor;

php	require_once('yocto_wakeupmonitor.php');
cpp	#include "yocto_wakeupmonitor.h"
m	#import "yocto_wakeupmonitor.h"
pas	uses yocto_wakeupmonitor;
vb	yocto_wakeupmonitor.vb
cs	yocto_wakeupmonitor.cs
java	import com.yoctopuce.YoctoAPI.YWakeUpMonitor;
py	from yocto_wakeupmonitor import *

Global functions

yFindWakeUpMonitor(func)

Retrieves a monitor for a given identifier.

yFirstWakeUpMonitor()

Starts the enumeration of monitors currently accessible.

`YWakeUpMonitor` methods

wakeupmonitor→describe()

Returns a short text that describes unambiguously the instance of the monitor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

wakeupmonitor→get_advertisedValue()

Returns the current value of the monitor (no more than 6 characters).

wakeupmonitor→get_errorMessage()

Returns the error message of the latest error with the monitor.

wakeupmonitor→get_errorType()

Returns the numerical error code of the latest error with the monitor.

wakeupmonitor→get_friendlyName()

Returns a global identifier of the monitor in the format `MODULE_NAME.FUNCTION_NAME`.

wakeupmonitor→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

wakeupmonitor→get_functionId()

Returns the hardware identifier of the monitor, without reference to the module.

wakeupmonitor→get_hardwareId()

Returns the unique hardware identifier of the monitor in the form `SERIAL.FUNCTIONID`.

wakeupmonitor→get_logicalName()

Returns the logical name of the monitor.

wakeupmonitor→get_module()

Gets the `YModule` object for the device on which the function is located.

wakeupmonitor→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

wakeupmonitor→get_nextWakeUp()

Returns the next scheduled wake up date/time (UNIX format)

wakeupmonitor→get_powerDuration()

Returns the maximal wake up time (in seconds) before automatically going to sleep.

wakeupmonitor→get_sleepCountdown()

Returns the delay before the next sleep period.

wakeupmonitor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

wakeupmonitor→get_wakeUpReason()

Returns the latest wake up reason.

wakeupmonitor→get_wakeUpState()

Returns the current state of the monitor

wakeupmonitor→isOnline()

Checks if the monitor is currently reachable, without raising any error.

wakeupmonitor→isOnline_async(callback, context)

Checks if the monitor is currently reachable, without raising any error (asynchronous version).

wakeupmonitor→load(msValidity)

Preloads the monitor cache with a specified validity duration.

wakeupmonitor→load_async(msValidity, callback, context)

Preloads the monitor cache with a specified validity duration (asynchronous version).

wakeupmonitor→nextWakeUpMonitor()

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

wakeupmonitor→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

wakeupmonitor→resetSleepCountDown()

Resets the sleep countdown.

wakeupmonitor→set_logicalName(newval)

Changes the logical name of the monitor.

wakeupmonitor→set_nextWakeUp(newval)

Changes the days of the week when a wake up must take place.

wakeupmonitor→set_powerDuration(newval)

Changes the maximal wake up time (seconds) before automatically going to sleep.

wakeupmonitor→set_sleepCountdown(newval)

Changes the delay before the next sleep period.

wakeupmonitor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

wakeupmonitor→sleep(secBeforeSleep)

Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.

wakeupmonitor→sleepFor(secUntilWakeUp, secBeforeSleep)

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.

wakeupmonitor→sleepUntil(wakeUpTime, secBeforeSleep)

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.

wakeupmonitor→wait_async(callback, context)

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()yFindWakeUpMonitor() YWakeUpMonitor

Retrieves a monitor for a given identifier.

```
function yFindWakeUpMonitor( 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()yFirstWakeUpMonitor() YWakeUpMonitor

Starts the enumeration of monitors currently accessible.

```
function yFirstWakeUpMonitor( )
```

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`.

```
function 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 monitor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

wakeupmonitor→get_advertisedValue() wakeupmonitor→advertisedValue() wakeupmonitor.get_advertisedValue() YWakeUpMonitor

Returns the current value of the monitor (no more than 6 characters).

function **get_advertisedValue**( )

Returns :

a string corresponding to the current value of the monitor (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

wakeupmonitor→get_errorMessage() wakeupmonitor→errorMessage() wakeupmonitor.get_errorMessage() YWakeUpMonitor

Returns the error message of the latest error with the monitor.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occured while using the monitor object

wakeupmonitor→get_errorType()
wakeupmonitor→errorType()
wakeupmonitor.get_errorType()

YWakeUpMonitor

Returns the numerical error code of the latest error with the monitor.

```
function 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 occured while using the monitor object

wakeupmonitor→get_friendlyName() wakeupmonitor→friendlyName() wakeupmonitor.get_friendlyName() YWakeUpMonitor

Returns a global identifier of the monitor in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the monitor if they are defined, otherwise the serial number of the module and the hardware identifier of the monitor (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the monitor using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

wakeupmonitor→get_functionDescriptor()
wakeupmonitor→functionDescriptor()
wakeupmonitor.get_functionDescriptor() YWakeUpMonitor

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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.

```
function get_functionId( )
```

For example `relay1`

Returns :

a string that identifies the monitor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

wakeupmonitor→get_hardwareId() wakeupmonitor→hardwareId() wakeupmonitor.get_hardwareId()

YWakeUpMonitor

Returns the unique hardware identifier of the monitor in the form `SERIAL.FUNCTIONID`.

```
function 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.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the monitor. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

wakeupmonitor→get_module() wakeupmonitor→module() wakeupmonitor.get_module() — YWakeUpMonitor

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() wakeupmonitor→module_async() wakeupmonitor.get_module_async()

YWakeUpMonitor

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

wakeupmonitor→get_nextWakeUp() wakeupmonitor→nextWakeUp() wakeupmonitor.get_nextWakeUp() YWakeUpMonitor

Returns the next scheduled wake up date/time (UNIX format)

function **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() wakeupmonitor→powerDuration() wakeupmonitor.get_powerDuration() YWakeUpMonitor

Returns the maximal wake up time (in seconds) before automatically going to sleep.

```
function get_powerDuration( )
```

Returns :

an integer corresponding to the maximal wake up time (in seconds) before automatically going to sleep

On failure, throws an exception or returns `Y_POWERDURATION_INVALID`.

wakeupmonitor→get_sleepCountdown() wakeupmonitor→sleepCountdown() wakeupmonitor.get_sleepCountdown() YWakeUpMonitor

Returns the delay before the next sleep period.

```
function get_sleepCountdown( )
```

Returns :

an integer corresponding to the delay before the next sleep period

On failure, throws an exception or returns `Y_SLEEPCOUNTDOWN_INVALID`.

wakeupmonitor→get_userData() wakeupmonitor→userData() wakeupmonitor.get_userData() YWakeUpMonitor

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

wakeupmonitor→get_wakeUpReason() wakeupmonitor→wakeUpReason() wakeupmonitor.get_wakeUpReason()

YWakeUpMonitor

Returns the latest wake up reason.

function **get_wakeUpReason**( )

Returns :

a value among Y_WAKEUPREASON_USBPOWER, Y_WAKEUPREASON_EXTPOWER, Y_WAKEUPREASON_ENDOFSLEEP, Y_WAKEUPREASON_EXTSIG1, Y_WAKEUPREASON_EXTSIG2, Y_WAKEUPREASON_EXTSIG3, Y_WAKEUPREASON_EXTSIG4, Y_WAKEUPREASON_SCHEDULE1, Y_WAKEUPREASON_SCHEDULE2, Y_WAKEUPREASON_SCHEDULE3, Y_WAKEUPREASON_SCHEDULE4, Y_WAKEUPREASON_SCHEDULE5 and Y_WAKEUPREASON_SCHEDULE6 corresponding to the latest wake up reason

On failure, throws an exception or returns Y_WAKEUPREASON_INVALID.

wakeupmonitor→get_wakeUpState() wakeupmonitor→wakeUpState() wakeupmonitor.get_wakeUpState()

YWakeUpMonitor

Returns the current state of the monitor

```
function 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.

```
function 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→isOnline_async() wakeupmonitor.isOnline_async() YWakeUpMonitor

Checks if the monitor is currently reachable, without raising any error (asynchronous version).

function **isOnline_async**(**callback**, **context**)

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

wakeupmonitor→load()wakeupmonitor.load() YWakeUpMonitor

Preloads the monitor cache with a specified validity duration.

```
function 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 trafic 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→load_async() wakeupmonitor.load_async() YWakeUpMonitor

Preloads the monitor cache with a specified validity duration (asynchronous version).

```
function load_async( msValidity, callback, context)
```

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

wakeupmonitor→nextWakeUpMonitor() wakeupmonitor.nextWakeUpMonitor() YWakeUpMonitor

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

```
function 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.

```
function 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.

```
function resetSleepCountDown( )
```

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_logicalName() wakeupmonitor→setLogicalName() wakeupmonitor.set_logicalName()

YWakeUpMonitor

Changes the logical name of the monitor.

```
function set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the monitor.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_nextWakeUp() wakeupmonitor→setNextWakeUp() wakeupmonitor.set_nextWakeUp()

YWakeUpMonitor

Changes the days of the week when a wake up must take place.

```
function set_nextWakeUp( newval)
```

Parameters :

newval an integer corresponding to the days of the week when a wake up must take place

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_powerDuration() wakeupmonitor→setPowerDuration() wakeupmonitor.set_powerDuration() YWakeUpMonitor

Changes the maximal wake up time (seconds) before automatically going to sleep.

```
function 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.

```
function set_sleepCountdown( newval)
```

Parameters :

newval an integer corresponding to the delay before the next sleep period

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_userData() wakeupmonitor→setUserData() wakeupmonitor.set_userData() YWakeUpMonitor

Stores a user context provided as argument in the userData attribute of the function.

```
function 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.

```
function 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.

```
function sleepFor( secUntilWakeUp, secBeforeSleep)
```

The count down before sleep can be canceled with resetSleepCountDown.

Parameters :

secUntilWakeUp	sleep duration, in secondes
secBeforeSleep	number of seconds before going into sleep mode

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→sleepUntil() wakeupmonitor.sleepUntil() YWakeUpMonitor

Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.

```
function 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→wait_async() wakeupmonitor.wait_async() YWakeUpMonitor

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

wakeupmonitor→wakeUp()wakeupmonitor.wakeUp() YWakeUpMonitor

Forces a wake up.

```
function wakeUp( )
```

3.44. WakeUpSchedule function interface

The WakeUpSchedule function implements a wake up condition. The wake up time is specified as a set of months and/or days and/or hours and/or minutes when the wake up should happen.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wakeupschedule.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YWakeUpSchedule = yoctolib.YWakeUpSchedule;

php	require_once('yocto_wakeupschedule.php');
cpp	#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 intervalof 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 occurence

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.

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()yFindWakeUpSchedule() YWakeUpSchedule

Retrieves a wake up schedule for a given identifier.

```
function yFindWakeUpSchedule( 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()yFirstWakeUpSchedule() YWakeUpSchedule

Starts the enumeration of wake up schedules currently accessible.

```
function yFirstWakeUpSchedule( )
```

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`.

```
function 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 wake up schedule (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

wakeupschedule→get_advertisedValue() wakeupschedule→advertisedValue() wakeupschedule.get_advertisedValue() YWakeUpSchedule

Returns the current value of the wake up schedule (no more than 6 characters).

```
function 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()
wakeupschedule→errorMessage()
wakeupschedule.get_errorMessage() YWakeUpSchedule

Returns the error message of the latest error with the wake up schedule.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the wake up schedule object

wakeupschedule→get_errorType() wakeupschedule→errorType() wakeupschedule.get_errorType() YWakeUpSchedule

Returns the numerical error code of the latest error with the wake up schedule.

```
function 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 occured while using the wake up schedule object

wakeupschedule→get_friendlyName() wakeupschedule→friendlyName() wakeupschedule.get_friendlyName()

YWakeUpSchedule

Returns a global identifier of the wake up schedule in the format `MODULE_NAME.FUNCTION_NAME`.

function **get_friendlyName**( )

The returned string uses the logical names of the module and of the wake up schedule if they are defined, otherwise the serial number of the module and the hardware identifier of the wake up schedule (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the wake up schedule using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

wakeupschedule→get_functionDescriptor() wakeupschedule→functionDescriptor() wakeupschedule.get_functionDescriptor() YWakeUpSchedule

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function get_functionDescriptor( )
```

This identifier can be used to test if two instances of `YFunction` reference the same physical function on the same physical device.

Returns :

an identifier of type `YFUN_DESCR`. If the function has never been contacted, the returned value is `Y_FUNCTIONDESCRIPTOR_INVALID`.

wakeupschedule→get_functionId() wakeupschedule→functionId() wakeupschedule.get_functionId() YWakeUpSchedule

Returns the hardware identifier of the wake up schedule, without reference to the module.

```
function get_functionId( )
```

For example `relay1`

Returns :

a string that identifies the wake up schedule (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

wakeupschedule→get_hardwareId() wakeupschedule→hardwareId() wakeupschedule.get_hardwareId() YWakeUpSchedule

Returns the unique hardware identifier of the wake up schedule in the form `SERIAL.FUNCTIONID`.

function **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()
wakeupschedule→hours()
wakeupschedule.get_hours()

YWakeUpSchedule

Returns the hours scheduled for wake up.

```
function get_hours( )
```

Returns :

an integer corresponding to the hours scheduled for wake up

On failure, throws an exception or returns `Y_HOURS_INVALID`.

wakeupschedule→get_logicalName() wakeupschedule→logicalName() wakeupschedule.get_logicalName() YWakeUpSchedule

Returns the logical name of the wake up schedule.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the wake up schedule. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

wakeupschedule→get_minutes()
wakeupschedule→minutes()
wakeupschedule.get_minutes()

YWakeUpSchedule

Returns all the minutes of each hour that are scheduled for wake up.

```
function get_minutes( )
```

wakeupschedule→get_minutesA() wakeupschedule→minutesA() wakeupschedule.get_minutesA() YWakeUpSchedule

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

```
function get_minutesA( )
```

Returns :

an integer corresponding to the minutes in the 00-29 interval of each hour scheduled for wake up

On failure, throws an exception or returns `Y_MINUTESA_INVALID`.

wakeupschedule→get_minutesB() wakeupschedule→minutesB() wakeupschedule.get_minutesB()

YWakeUpSchedule

Returns the minutes in the 30-59 intervalof each hour scheduled for wake up.

```
function get_minutesB( )
```

Returns :

an integer corresponding to the minutes in the 30-59 intervalof each hour scheduled for wake up

On failure, throws an exception or returns `Y_MINUTESB_INVALID`.

wakeupschedule→get_module() wakeupschedule→module() wakeupschedule.get_module() YWakeUpSchedule

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() wakeupschedule→module_async() wakeupschedule.get_module_async()

YWakeUpSchedule

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

wakeupschedule→get_monthDays() wakeupschedule→monthDays() wakeupschedule.get_monthDays() YWakeUpSchedule

Returns the days of the month scheduled for wake up.

```
function 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()
wakeupschedule→months()
wakeupschedule.get_months()

YWakeUpSchedule

Returns the months scheduled for wake up.

```
function get_months( )
```

Returns :

an integer corresponding to the months scheduled for wake up

On failure, throws an exception or returns `Y_MONTHS_INVALID`.

wakeupschedule→get_nextOccurence() wakeupschedule→nextOccurence() wakeupschedule.get_nextOccurence() YWakeUpSchedule

Returns the date/time (seconds) of the next wake up occurence

```
function get_nextOccurence( )
```

Returns :

an integer corresponding to the date/time (seconds) of the next wake up occurence

On failure, throws an exception or returns `Y_NEXTOCCURENCE_INVALID`.

wakeupschedule→get_userData() wakeupschedule→userData() wakeupschedule.get_userData() YWakeUpSchedule

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

wakeupschedule→get_weekDays() wakeupschedule→weekDays() wakeupschedule.get_weekDays() YWakeUpSchedule

Returns the days of the week scheduled for wake up.

```
function 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.

```
function 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→isOnline_async() wakeupschedule.isOnline_async() YWakeUpSchedule

Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

wakeupschedule→load()wakeupschedule.load() YWakeUpSchedule

Preloads the wake up schedule cache with a specified validity duration.

```
function 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 trafic 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→load_async() wakeupschedule.load_async() — YWakeUpSchedule

Preloads the wake up schedule cache with a specified validity duration (asynchronous version).

function **load_async**(**msValidity**, **callback**, **context**)

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

wakeupschedule→nextWakeUpSchedule() wakeupschedule.nextWakeUpSchedule() YWakeUpSchedule

Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

function **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.

```
function registerValueCallback( callback)
```

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

wakeupschedule→set_hours() wakeupschedule→setHours() wakeupschedule.set_hours() YWakeUpSchedule

Changes the hours when a wake up must take place.

```
function set_hours( newval)
```

Parameters :

newval an integer corresponding to the hours when a wake up must take place

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_logicalName() wakeupschedule→setLogicalName() wakeupschedule.set_logicalName()

YWakeUpSchedule

Changes the logical name of the wake up schedule.

```
function set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the wake up schedule.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutes()
wakeupschedule→setMinutes()
wakeupschedule.set_minutes() YWakeUpSchedule

Changes all the minutes where a wake up must take place.

```
function set_minutes( bitmap)
```

Parameters :

bitmap Minutes 00-59 of each hour scheduled for wake up.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutesA() wakeupschedule→setMinutesA() wakeupschedule.set_minutesA()

YWakeUpSchedule

Changes the minutes in the 00-29 interval when a wake up must take place.

```
function set_minutesA( newval)
```

Parameters :

newval an integer corresponding to the minutes in the 00-29 interval when a wake up must take place

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutesB() wakeupschedule→setMinutesB() wakeupschedule.set_minutesB() YWakeUpSchedule

Changes the minutes in the 30-59 interval when a wake up must take place.

```
function set_minutesB( newval)
```

Parameters :

newval an integer corresponding to the minutes in the 30-59 interval when a wake up must take place

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_monthDays() wakeupschedule→setMonthDays() wakeupschedule.set_monthDays() YWakeUpSchedule

Changes the days of the month when a wake up must take place.

```
function set_monthDays( newval)
```

Parameters :

newval an integer corresponding to the days of the month when a wake up must take place

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_months() wakeupschedule→setMonths() wakeupschedule.set_months()

YWakeUpSchedule

Changes the months when a wake up must take place.

```
function set_months( newval)
```

Parameters :

newval an integer corresponding to the months when a wake up must take place

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_userData() wakeupschedule→setUserData() wakeupschedule.set_userData() YWakeUpSchedule

Stores a user context provided as argument in the userData attribute of the function.

```
function 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() wakeupschedule→setWeekDays() wakeupschedule.set_weekDays() YWakeUpSchedule

Changes the days of the week when a wake up must take place.

```
function 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.

wakeupschedule→wait_async() wakeupschedule.wait_async() YWakeUpSchedule

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.45. Watchdog function interface

The watchog 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 direcly 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 runing 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 runningsttae 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()yFindWatchdog() YWatchdog

Retrieves a watchdog for a given identifier.

```
function yFindWatchdog( 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()yFirstWatchdog() YWatchdog

Starts the enumeration of watchdog currently accessible.

```
function yFirstWatchdog( )
```

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.

```
function delayedPulse( ms_delay, ms_duration)
```

Parameters :

ms_delay	waiting time before the pulse, in millisecondes
ms_duration	pulse duration, in millisecondes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→describe()watchdog.describe() YWatchdog

Returns a short text that describes unambiguously the instance of the watchdog in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function 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 watchdog (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

watchdog→get_advertisedValue() watchdog→advertisedValue() watchdog.get_advertisedValue() YWatchdog

Returns the current value of the watchdog (no more than 6 characters).

```
function 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() watchdog→autoStart()watchdog.get_autoStart() YWatchdog

Returns the watchdog runing state at module power on.

```
function get_autoStart( )
```

Returns :

either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the watchdog runing state at module power on

On failure, throws an exception or returns `Y_AUTOSTART_INVALID`.

watchdog→get_countdown() watchdog→countdown()watchdog.get_countdown() YWatchdog

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

function **get_countdown**( )

Returns :

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns `Y_COUNTDOWN_INVALID`.

watchdog→get_errorMessage()
watchdog→errorMessage()
watchdog.get_errorMessage()

YWatchdog

Returns the error message of the latest error with the watchdog.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occured while using the watchdog object

watchdog→get_errorType() watchdog→errorType()watchdog.get_errorType() YWatchdog

Returns the numerical error code of the latest error with the watchdog.

```
function 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 occured while using the watchdog object

watchdog→get_friendlyName() watchdog→friendlyName() watchdog.get_friendlyName() YWatchdog

Returns a global identifier of the watchdog in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the watchdog if they are defined, otherwise the serial number of the module and the hardware identifier of the watchdog (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the watchdog using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

watchdog→get_functionDescriptor() watchdog→functionDescriptor() watchdog.get_functionDescriptor() YWatchdog

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

function **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() watchdog→functionId()watchdog.get_functionId() YWatchdog

Returns the hardware identifier of the watchdog, without reference to the module.

```
function 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() watchdog→hardwareId()watchdog.get_hardwareId() YWatchdog

Returns the unique hardware identifier of the watchdog in the form `SERIAL.FUNCTIONID`.

```
function get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the watchdog. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the watchdog (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

watchdog→get_logicalName() watchdog→logicalName() watchdog.get_logicalName() YWatchdog

Returns the logical name of the watchdog.

```
function get_logicalName( )
```

Returns :

a string corresponding to the logical name of the watchdog. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

watchdog→get_maxTimeOnStateA() watchdog→maxTimeOnStateA() watchdog.get_maxTimeOnStateA() YWatchdog

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

```
function get_maxTimeOnStateA( )
```

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEA_INVALID`.

watchdog→get_maxTimeOnStateB() watchdog→maxTimeOnStateB() watchdog.get_maxTimeOnStateB() YWatchdog

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

```
function get_maxTimeOnStateB( )
```

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEB_INVALID`.

watchdog→get_module() watchdog→module()watchdog.get_module() YWatchdog

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() watchdog→module_async() watchdog.get_module_async() YWatchdog

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

watchdog→get_output() watchdog→output()watchdog.get_output() YWatchdog

Returns the output state of the watchdog, when used as a simple switch (single throw).

```
function 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() watchdog→pulseTimer()watchdog.get_pulseTimer() YWatchdog

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

```
function 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() watchdog→running()watchdog.get_running() YWatchdog

Returns the watchdog running state.

```
function 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() watchdog→state()watchdog.get_state() YWatchdog

Returns the state of the watchdog (A for the idle position, B for the active position).

```
function get_state( )
```

Returns :

either `Y_STATE_A` or `Y_STATE_B`, according to the state of the watchdog (A for the idle position, B for the active position)

On failure, throws an exception or returns `Y_STATE_INVALID`.

watchdog→get_stateAtPowerOn() watchdog→stateAtPowerOn() watchdog.get_stateAtPowerOn() — YWatchdog

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

```
function get_stateAtPowerOn( )
```

Returns :

a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B` corresponding to the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns `Y_STATEATPOWERON_INVALID`.

watchdog→get_triggerDelay() watchdog→triggerDelay() watchdog.get_triggerDelay() — YWatchdog

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

```
function get_triggerDelay( )
```

Returns :

an integer corresponding to the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds

On failure, throws an exception or returns `Y_TRIGGERDELAY_INVALID`.

watchdog→get_triggerDuration() watchdog→triggerDuration() watchdog.get_triggerDuration() YWatchdog

Returns the duration of resets caused by the watchdog, in milliseconds.

```
function 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() watchdog→userData()watchdog.get_userData() YWatchdog

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function 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.

```
function 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→isOnline_async() watchdog.isOnline_async() YWatchdog

Checks if the watchdog is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

watchdog→load()watchdog.load() YWatchdog

Preloads the watchdog cache with a specified validity duration.

```
function 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 trafic 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→load_async()watchdog.load_async() YWatchdog

Preloads the watchdog cache with a specified validity duration (asynchronous version).

function **load_async**(**msValidity**, **callback**, **context**)

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

watchdog→nextWatchdog() watchdog.nextWatchdog() YWatchdog

Continues the enumeration of watchdog started using `yFirstWatchdog()`.

```
function 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).

```
function pulse( ms_duration)
```

Parameters :

ms_duration pulse duration, in millisecondes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→registerValueCallback() watchdog.registerValueCallback() YWatchdog

Registers the callback function that is invoked on every change of advertised value.

```
function 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.

```
function resetWatchdog( )
```

When the watchdog is running, this function must be called on a regular basis to prevent the watchog to trigger

Returns :
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_autoStart() watchdog→setAutoStart()watchdog.set_autoStart() YWatchdog

Changes the watchdog runningsttae at module power on.

```
function set_autoStart( newval)
```

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the watchdog runningsttae at module power on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_logicalName()
watchdog→setLogicalName()
watchdog.set_logicalName() YWatchdog

Changes the logical name of the watchdog.

```
function set_logicalName( newval)
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the watchdog.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.

watchdog→set_maxTimeOnStateA() watchdog→setMaxTimeOnStateA() watchdog.set_maxTimeOnStateA() YWatchdog

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

```
function set_maxTimeOnStateA( newval)
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_maxTimeOnStateB() watchdog→setMaxTimeOnStateB() watchdog.set_maxTimeOnStateB() YWatchdog

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

```
function 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() watchdog→setOutput()watchdog.set_output() YWatchdog

Changes the output state of the watchdog, when used as a simple switch (single throw).

```
function 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() watchdog→setRunning()watchdog.set_running() YWatchdog

Changes the running state of the watchdog.

```
function 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() watchdog→setState()watchdog.set_state() YWatchdog

Changes the state of the watchdog (A for the idle position, B for the active position).

```
function set_state( newval)
```

Parameters :

newval either `Y_STATE_A` or `Y_STATE_B`, according to the state of the watchdog (A for the idle position, B for the active position)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_stateAtPowerOn()
watchdog→setStateAtPowerOn()
watchdog.set_stateAtPowerOn() YWatchdog

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

```
function 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.

```
function 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.

```
function 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() watchdog→setUserData()watchdog.set_userData() YWatchdog

Stores a user context provided as argument in the userData attribute of the function.

```
function 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

watchdog→wait_async()watchdog.wait_async() YWatchdog

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

3.46. Wireless function interface

YWireless functions provides control over wireless network parameters and status for devices that are wireless-enabled.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_wireless.js'></script>

nodejs	var yoctolib = require('yoctolib');
	var YWireless = yoctolib.YWireless;

php	require_once('yocto_wireless.php');
cpp	#include "yocto_wireless.h"
m	#import "yocto_wireless.h"
pas	uses yocto_wireless;
vb	yocto_wireless.vb
cs	yocto_wireless.cs
java	import com.yoctopuce.YoctoAPI.YWireless;
py	from yocto_wireless import *

Global functions

yFindWireless(**func**)

Retrieves a wireless lan interface for a given identifier.

yFirstWireless()

Starts the enumeration of wireless lan interfaces currently accessible.

`YWireless` methods

wireless→adhocNetwork(**ssid**, **securityKey**)

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

wireless→describe()

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

wireless→get_advertisedValue()

Returns the current value of the wireless lan interface (no more than 6 characters).

wireless→get_channel()

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

wireless→get_detectedWlans()

Returns a list of YWlanRecord objects that describe detected Wireless networks.

wireless→get_errorMessage()

Returns the error message of the latest error with the wireless lan interface.

wireless→get_errorType()

Returns the numerical error code of the latest error with the wireless lan interface.

wireless→get_friendlyName()

Returns a global identifier of the wireless lan interface in the format `MODULE_NAME.FUNCTION_NAME`.

wireless→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

wireless→get_functionId()

Returns the hardware identifier of the wireless lan interface, without reference to the module.

wireless→get_hardwareId()

Returns the unique hardware identifier of the wireless lan interface in the form `SERIAL.FUNCTIONID`.

wireless→get_linkQuality()
Returns the link quality, expressed in percent.

wireless→get_logicalName()
Returns the logical name of the wireless lan interface.

wireless→get_message()
Returns the latest status message from the wireless interface.

wireless→get_module()
Gets the `YModule` object for the device on which the function is located.

wireless→get_module_async(callback, context)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

wireless→get_security()
Returns the security algorithm used by the selected wireless network.

wireless→get_ssid()
Returns the wireless network name (SSID).

wireless→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

wireless→isOnline()
Checks if the wireless lan interface is currently reachable, without raising any error.

wireless→isOnline_async(callback, context)
Checks if the wireless lan interface is currently reachable, without raising any error (asynchronous version).

wireless→joinNetwork(ssid, securityKey)
Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

wireless→load(msValidity)
Preloads the wireless lan interface cache with a specified validity duration.

wireless→load_async(msValidity, callback, context)
Preloads the wireless lan interface cache with a specified validity duration (asynchronous version).

wireless→nextWireless()
Continues the enumeration of wireless lan interfaces started using `yFirstWireless()`.

wireless→registerValueCallback(callback)
Registers the callback function that is invoked on every change of advertised value.

wireless→set_logicalName(newval)
Changes the logical name of the wireless lan interface.

wireless→set_userData(data)
Stores a user context provided as argument in the userData attribute of the function.

wireless→wait_async(callback, context)
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YWireless.FindWireless() yFindWireless()yFindWireless() — YWireless

Retrieves a wireless lan interface for a given identifier.

```
function yFindWireless( 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()yFirstWireless() YWireless

Starts the enumeration of wireless lan interfaces currently accessible.

```
function yFirstWireless( )
```

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.

```
function adhocNetwork( ssid, securityKey)
```

If a security key is specified, the network is protected by WEP128, since WPA is not standardized for ad-hoc networks. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid	the name of the network to connect to
securityKey	the network key, as a character string

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→describe()wireless.describe() YWireless

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
function 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() wireless→advertisedValue() wireless.get_advertisedValue() YWireless

Returns the current value of the wireless lan interface (no more than 6 characters).

```
function 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() wireless→channel()wireless.get_channel() YWireless

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

```
function get_channel( )
```

Returns :

an integer corresponding to the 802.11 channel currently used, or 0 when the selected network has not been found

On failure, throws an exception or returns `Y_CHANNEL_INVALID`.

wireless→get_detectedWlans() wireless→detectedWlans() wireless.get_detectedWlans() YWireless

Returns a list of YWlanRecord objects that describe detected Wireless networks.

```
function get_detectedWlans( )
```

This list is not updated when the module is already connected to an acces point (infrastructure mode). To force an update of this list, `adhocNetwork()` must be called to disconnect the module from the current network. The returned list must be unallocated by the caller.

Returns :
a list of `YWlanRecord` objects, containing the SSID, channel, link quality and the type of security of the wireless network.

On failure, throws an exception or returns an empty list.

wireless→get_errorMessage() wireless→errorMessage() wireless.get_errorMessage() YWireless

Returns the error message of the latest error with the wireless lan interface.

```
function get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the wireless lan interface object

wireless→get_errorType() wireless→errorType()wireless.get_errorType() YWireless

Returns the numerical error code of the latest error with the wireless lan interface.

```
function 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 occured while using the wireless lan interface object

wireless→get_friendlyName() wireless→friendlyName()wireless.get_friendlyName() YWireless

Returns a global identifier of the wireless lan interface in the format `MODULE_NAME.FUNCTION_NAME`.

```
function get_friendlyName( )
```

The returned string uses the logical names of the module and of the wireless lan interface if they are defined, otherwise the serial number of the module and the hardware identifier of the wireless lan interface (for exemple: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the wireless lan interface using logical names (ex: `MyCustomName.relay1`) On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

wireless→get_functionDescriptor() wireless→functionDescriptor() wireless.get_functionDescriptor() YWireless

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

```
function 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() wireless→functionId()wireless.get_functionId() YWireless

Returns the hardware identifier of the wireless lan interface, without reference to the module.

```
function 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() wireless→hardwareId()wireless.get_hardwareId() YWireless

Returns the unique hardware identifier of the wireless lan interface in the form `SERIAL.FUNCTIONID`.

```
function 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() wireless→linkQuality()wireless.get_linkQuality() YWireless

Returns the link quality, expressed in percent.

```
function 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() wireless→logicalName()wireless.get_logicalName() YWireless

Returns the logical name of the wireless lan interface.

function **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() wireless→message()wireless.get_message() YWireless

Returns the latest status message from the wireless interface.

```
function 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() wireless→module()wireless.get_module() YWireless

Gets the `YModule` object for the device on which the function is located.

```
function 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_module_async() wireless→module_async() wireless.get_module_async() YWireless

Gets the `YModule` object for the device on which the function is located (asynchronous version).

function **get_module_async**(**callback**, **context**)

If the function cannot be located on any module, the returned `YModule` object does not show as on-line. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

wireless→get_security() wireless→security()wireless.get_security() YWireless

Returns the security algorithm used by the selected wireless network.

```
function 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() wireless→ssid()wireless.get_ssid() YWireless

Returns the wireless network name (SSID).

```
function 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() wireless→userData()wireless.get_userData() YWireless

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
function 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.

```
function 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→isOnline_async()wireless.isOnline_async() YWireless

Checks if the wireless lan interface is currently reachable, without raising any error (asynchronous version).

```
function isOnline_async( callback, context)
```

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 requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

wireless→joinNetwork()wireless.joinNetwork() YWireless

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

```
function joinNetwork( ssid, securityKey)
```

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid	the name of the network to connect to
securityKey	the network key, as a character string

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→load()wireless.load() YWireless

Preloads the wireless lan interface cache with a specified validity duration.

```
function 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 trafic 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→load_async()wireless.load_async() YWireless

Preloads the wireless lan interface cache with a specified validity duration (asynchronous version).

function **load_async**(**msValidity**, **callback**, **context**)

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 trafic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

wireless→nextWireless()wireless.nextWireless() YWireless

Continues the enumeration of wireless lan interfaces started using `yFirstWireless()`.

```
function 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.

```
function 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.

```
function 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() wireless→setUserData()wireless.set_userData() YWireless

Stores a user context provided as argument in the userData attribute of the function.

```
function 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→wait_async()wireless.wait_async() YWireless

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback	callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing.

Index