



# JavaScript API Reference

# Table of contents

<b>Introduction</b>	<b>3</b>
<b>Using Yocto—Demo with Javascript</b>	<b>4</b>
Getting ready	4
Control of the Led function	4
Control of the module part	6
Error handling	9
<b>Reference</b>	<b>10</b>
General functions	10
AnButton function interface	16
CarbonDioxide function interface	25
ColorLed function interface	34
Current function interface	42
DataLogger function interface	50
Formatted data sequence	59
Recorded data sequence	62
External power supply control interface	65
Yocto-hub port interface	72
Humidity function interface	79
Led function interface	87
LightSensor function interface	94
Module control interface	103
Network function interface	114
Pressure function interface	127
Relay function interface	135
Servo function interface	143
Temperature function interface	150
Voltage function interface	159
Voltage source function interface	168
Wireless function interface	177
<b>Index</b>	<b>185</b>

# 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<sup>1</sup>, 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<sup>2</sup>
- The VirtualHub software<sup>3</sup> 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 snippet 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())
{
    // Use led.set_power(), ...
}
```

Let's look at these lines in more details.

---

<sup>1</sup> Actually, as soon as Opera implements support for the HTTP Access-Control-Allow-Origin header.

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

<sup>3</sup> [www.yoctopuce.com/EN/virtualhub.php](http://www.yoctopuce.com/EN/virtualhub.php)

## 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 object 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<sup>4</sup>, 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 last 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.

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>
```

---

<sup>4</sup> If you do not have a text editor, use Notepad rather than Microsoft Word.

```

<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 connected 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 connected 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>';
    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:

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

#### 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.

##### `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.

**yRegisterLogFunction(logfun)**

Register a log callback function.

**ySetDelegate(object)**

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

**ySetTimeout(callback, ms\_timeout, optional\_arguments)**

Invoke the specified callback function after a given timeout.

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

Pauses the execution flow for a specified duration.

**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.

## yCheckLogicalName()

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.

## yDisableExceptions()

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.

## yEnableExceptions()

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

function **yEnableExceptions( )**

Be aware that 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.

---

This function is used only on Android.

Before calling `yRegisterHub("usb")` you need to activate the USB host port of the system. This function takes as argument, an object of class `android.content.Context` (or any subclass). It is not necessary to call this function to reach modules through the network.

**Parameters :**

**osContext** an object of class `android.content.Context` (or any subclass).  
On failure, throws an exception.

---

## **yFreeAPI()**

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.

---

## **yGetAPIVersion()**

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.

---

## **yGetTickCount()**

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.

---

## **yHandleEvents()**

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.

---

## **yInitAPI()**

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.

---

## **yRegisterDeviceArrivalCallback()**

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` to unregister a previously registered callback.

---

## **yRegisterDeviceRemovalCallback()**

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` to unregister a previously registered callback.

---

## yRegisterHub()

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

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

When using Yoctopuce modules through the VirtualHub gateway, you should provide as parameter the address of the machine on which the VirtualHub software is running (typically "http://127.0.0.1:4444", which represents the local machine). When you use a language which has direct access to the USB hardware, you can use the pseudo-URL "usb" instead.

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 VirtualHub you want to reach, the URL parameter should look like: `http://username:password@adresse:port`

### Parameters :

- url** a string containing either "usb" 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.

---

Register a log callback function.

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

### Parameters :

- logfun** a procedure taking a string parameter, or null to unregister a previously registered callback.

---

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

The methodes `yDeviceArrival` and `yDeviceRemoval` will be invoked while `yUpdateDeviceList` is running. You will have to call this function on a regular basis.

### Parameters :

- object** an object that must follow the protocol YAPIDelegate, or nil to unregister a previously registered object.

---

## ySetTimeout()

Invoke the specified callback function after a given timeout.

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

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.  
**optional\_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.

---

## ySleep()

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.

---

## yUnregisterHub()

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 root URL of the hub to monitor

## yUpdateDeviceList()

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.

---

## yUpdateDeviceList\_async()

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 two arguments: the caller-specific context object and the result code (`YAPI_SUCCESS` if the operation completes successfully).

**context** caller-specific object that is passed as-is to the callback function

### Returns :

nothing : the result is provided to the callback.

## 3.2. 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:

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

### 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 descriptive text that identifies the function.

#### 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 this function.

**anbutton→get\_errorType()**

Returns the numerical error code of the latest error with this function.

**anbutton→get\_functionDescriptor()**

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

**anbutton→get\_hardwareId()**

Returns the unique hardware identifier of the function.

**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()**

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

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

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

**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 255, included) 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 function is currently reachable, without raising any error.

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

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

**anbutton→load(msValidity)**

Preloads the function cache with a specified validity duration.

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

Preloads the function 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→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 255, included) for triggering user callbacks.

**anbutton→set\_userData(data)**

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

## yFindAnButton()

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.

## yFirstAnButton()

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()

Returns a descriptive text that identifies the function.

function **describe( )**

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

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

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()**

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()**

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()**

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()**

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()**

Returns the error message of the latest error with this function.

---

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 occurred while using this function object

---

**anbutton.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

**anbutton.get\_anbuttonDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function **get\_hardwareId( )**

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function. On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

**anbutton.get\_isPressed()**

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()**

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

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()**

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_LASTTIMERELASED_INVALID`.

---

**anbutton.get\_logicalName()**

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()**

Get 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()**

Get 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\_rawValue()**

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()**

Returns the sensibility for the input (between 1 and 255, included) for triggering user callbacks.

function **get\_sensitivity**( )

**Returns :**  
an integer corresponding to the sensibility for the input (between 1 and 255, included) for triggering user callbacks  
On failure, throws an exception or returns `Y_SENSITIVITY_INVALID`.

---

### **anbutton.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.

---

### **anbutton.isOnline()**

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

function **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

---

### **anbutton.isOnline\_async()**

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.

---

## **anbutton.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

## **anbutton.load\_async()**

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 traffic 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.

---

## **anbutton.nextAnButton()**

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()**

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.set\_analogCalibration()**

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()**

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()**

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()**

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()**

Changes the sensibility for the input (between 1 and 255, included) for triggering user callbacks.

function **set\_sensitivity**( **newval**)

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 255, included) for triggering user callbacks

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

### **anbutton.set\_userData()**

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

## **3.3. 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:

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

## 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 descriptive text that identifies the function.

### 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 measured value.

### carbondioxide→get\_errorMessage()

Returns the error message of the latest error with this function.

### carbondioxide→get\_errorType()

Returns the numerical error code of the latest error with this function.

### carbondioxide→get\_functionDescriptor()

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

### carbondioxide→get\_hardwareId()

Returns the unique hardware identifier of the function.

### carbondioxide→get\_highestValue()

Returns the maximal value observed.

### carbondioxide→get\_logicalName()

Returns the logical name of the CO2 sensor.

### carbondioxide→get\_lowestValue()

Returns the minimal value observed.

### carbondioxide→get\_module()

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

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

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

### carbondioxide→get\_resolution()

Returns the resolution of the measured values.

### carbondioxide→get\_unit()

Returns the measuring unit for the measured value.

### carbondioxide→get\_userData()

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

### carbondioxide→isOnline()

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

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

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

**carbondioxide→load(msValidity)**

Preloads the function cache with a specified validity duration.

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

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

**carbondioxide→nextCarbonDioxide()**

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

**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\_logicalName(newval)**

Changes the logical name of the CO2 sensor.

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

Changes the recorded minimal value observed.

**carbondioxide→set\_userData(data)**

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

## yFindCarbonDioxide()

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.

## yFirstCarbonDioxide()

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()**

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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

**carbondioxide.describe()**

Returns a descriptive text that identifies the function.

```
function describe( )
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

**carbondioxide.get\_advertisedValue()**

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()**

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()**

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`.

---

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

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

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

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

## **carbondioxide.get\_carbondioxideDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function **get\_hardwareId**( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

### **Returns :**

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

## **carbondioxide.get\_highestValue()**

Returns the maximal value observed.

function **get\_highestValue**( )

**Returns :**

a floating point number corresponding to the maximal value observed

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

---

**carbondioxide.get\_logicalName()**

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()**

Returns the minimal value observed.

function **get\_lowestValue**( )

**Returns :**

a floating point number corresponding to the minimal value observed

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

---

**carbondioxide.get\_module()**

Get 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()**

Get 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\_resolution()**

Returns the resolution of the measured values.

function **get\_resolution**( )

The resolution corresponds to the numerical precision of the values, 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()**

Returns the measuring unit for the measured value.

function **get\_unit**( )

### **Returns :**

a string corresponding to the measuring unit for the measured value

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

---

## **carbondioxide.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.

---

## **carbondioxide.isOnline()**

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

function **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

---

## **carbondioxide.isOnline\_async()**

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.

---

## **carbondioxide.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

## **carbondioxide.load\_async()**

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 traffic 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.

---

## **carbondioxide.nextCarbonDioxide()**

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.registerValueCallback()

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()

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\_logicalName()

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()

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\_userdata()

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

## 3.4. ColorLed function interface

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

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

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

### 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 descriptive text that identifies the function.

#### 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 this function.

#### colorled→get\_errorType()

Returns the numerical error code of the latest error with this function.

#### colorled→get\_functionDescriptor()

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

#### colorled→get\_hardwareId()

Returns the unique hardware identifier of the function.

#### 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()

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

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

Get 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.

<b>colored→get_rgbColorAtPowerOn()</b>
Returns the configured color to be displayed when the module is turned on.
<b>colored→get_userData()</b>
Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
<b>colored→hslMove(hsl_target, ms_duration)</b>
Performs a smooth transition in the HSL color space between the current color and a target color.
<b>colored→isOnline()</b>
Checks if the function is currently reachable, without raising any error.
<b>colored→isOnline_async(callback, context)</b>
Checks if the function is currently reachable, without raising any error (asynchronous version).
<b>colored→load(msValidity)</b>
Preloads the function cache with a specified validity duration.
<b>colored→load_async(msValidity, callback, context)</b>
Preloads the function cache with a specified validity duration (asynchronous version).
<b>colored→nextColorLed()</b>
Continues the enumeration of RGB leds started using <code>yFirstColorLed()</code> .
<b>colored→registerValueCallback(callback)</b>
Registers the callback function that is invoked on every change of advertised value.
<b>colored→rgbMove(rgb_target, ms_duration)</b>
Performs a smooth transition in the RGB color space between the current color and a target color.
<b>colored→set_hslColor(newval)</b>
Changes the current color of the led, using a color HSL.
<b>colored→set_logicalName(newval)</b>
Changes the logical name of the RGB led.
<b>colored→set_rgbColor(newval)</b>
Changes the current color of the led, using a RGB color.
<b>colored→set_rgbColorAtPowerOn(newval)</b>
Changes the color that the led will display by default when the module is turned on.
<b>colored→set_userData(data)</b>
Stores a user context provided as argument in the userData attribute of the function.

## yFindColorLed()

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.

---

## **yFirstColorLed()**

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()**

Returns a descriptive text that identifies the function.

function **describe( )**

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

## **colorled.get\_advertisedValue()**

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()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

## **colorled.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

**colorled.get\_colorledDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function **get\_hardwareId**( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function. On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

**colorled.get\_hslColor()**

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()**

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()**

Get 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()

Get 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()

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()

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()

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.hsIMove()

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

function **hsIMove**( **hsl\_target**, **ms\_duration**)

**Parameters :**

**hsl\_target** desired HSL color at the end of the transition  
**ms\_duration** duration of the transition, in millisecond

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

**colorled.isOnline()**

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

function **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

---

**colorled.isOnline\_async()**

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.

---

**colorled.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

---

## colorled.load\_async()

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 traffic 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.

---

## colorled.nextColorLed()

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()

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()

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()**

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()**

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()**

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()**

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.

---

## colored.set\_userData()

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

## 3.5. 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:

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

### 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 descriptive text that identifies the function.

#### 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 measured value.

#### current→get\_errorMessage()

Returns the error message of the latest error with this function.

#### current→get\_errorType()

Returns the numerical error code of the latest error with this function.

#### current→get\_functionDescriptor()

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

#### current→get\_hardwareId()

Returns the unique hardware identifier of the function.

<b>current→get_highestValue()</b>
Returns the maximal value observed.
<b>current→get_logicalName()</b>
Returns the logical name of the current sensor.
<b>current→get_lowestValue()</b>
Returns the minimal value observed.
<b>current→get_module()</b>
Get the <code>YModule</code> object for the device on which the function is located.
<b>current→get_module_async(callback, context)</b>
Get the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<b>current→get_resolution()</b>
Returns the resolution of the measured values.
<b>current→get_unit()</b>
Returns the measuring unit for the measured value.
<b>current→get_userData()</b>
Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<b>current→isOnline()</b>
Checks if the function is currently reachable, without raising any error.
<b>current→isOnline_async(callback, context)</b>
Checks if the function is currently reachable, without raising any error (asynchronous version).
<b>current→load(msValidity)</b>
Preloads the function cache with a specified validity duration.
<b>current→load_async(msValidity, callback, context)</b>
Preloads the function cache with a specified validity duration (asynchronous version).
<b>current→nextCurrent()</b>
Continues the enumeration of current sensors started using <code>yFirstCurrent()</code> .
<b>current→registerValueCallback(callback)</b>
Registers the callback function that is invoked on every change of advertised value.
<b>current→set_highestValue(newval)</b>
Changes the recorded maximal value observed.
<b>current→set_logicalName(newval)</b>
Changes the logical name of the current sensor.
<b>current→set_lowestValue(newval)</b>
Changes the recorded minimal value observed.
<b>current→set_userData(data)</b>
Stores a user context provided as argument in the <code>userData</code> attribute of the function.

## yFindCurrent()

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.

---

## **yFirstCurrent()**

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()**

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 lineat interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

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

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

## **current.describe()**

Returns a descriptive text that identifies the function.

function **describe()**

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

**current.get\_advertisedValue()**

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()**

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()**

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`.

---

**current.get\_errorMessage()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

**current.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

---

## **current.get\_currentDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function **get\_hardwareId()** ( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

## **current.get\_highestValue()**

Returns the maximal value observed.

function **get\_highestValue()** ( )

**Returns :**

a floating point number corresponding to the maximal value observed

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

---

## **current.get\_logicalName()**

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()**

Returns the minimal value observed.

function **get\_lowestValue()** ( )

**Returns :**

a floating point number corresponding to the minimal value observed

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

---

## **current.get\_module()**

Get 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()**

Get 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\_resolution()**

Returns the resolution of the measured values.

function **get\_resolution**( )

The resolution corresponds to the numerical precision of the values, 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()**

Returns the measuring unit for the measured value.

function **get\_unit**( )

**Returns :**  
a string corresponding to the measuring unit for the measured value  
On failure, throws an exception or returns `Y_UNIT_INVALID`.

---

### **current.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.

---

### **current.isOnline()**

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

function **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

---

### **current.isOnline\_async()**

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.

---

### **current.load()**

Preloads the function 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 traffic for instance.

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

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

---

### **current.load\_async()**

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 traffic 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.

---

## **current.nextCurrent()**

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.registerValueCallback()**

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()**

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.

---

## **current.set\_logicalName()**

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()**

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.

---

### **current.set\_userData()**

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

## **3.6. 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 Yoctopuce application programming interface includes functions to control how this internal data logger works. Because the sensors do not include a battery, they do not have an absolute time reference. Therefore, measures are simply indexed by the absolute run number and time relative to the start of the run. Every new power up starts a new run. It is however possible to setup an absolute UTC time by software at a given time, so that the data logger keeps track of it until it is next powered off.

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

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

#### **Global functions**

##### **yFindDataLogger(func)**

Retrieves a data logger for a given identifier.

##### **yFirstDataLogger()**

Starts the enumeration of data loggers currently accessible.

#### **YDataLogger methods**

<b><code>datalogger→describe()</code></b>	Returns a descriptive text that identifies the function.
<b><code>datalogger→forgetAllDataStreams()</code></b>	Clears the data logger memory and discards all recorded data streams.
<b><code>datalogger→get_advertisedValue()</code></b>	Returns the current value of the data logger (no more than 6 characters).
<b><code>datalogger→get_autoStart()</code></b>	Returns the default activation state of the data logger on power up.
<b><code>datalogger→get_currentRunIndex()</code></b>	Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.
<b><code>datalogger→get_dataRun(runIdx)</code></b>	Returns a data run object holding all measured data for a given period during which the module was turned on (a run).
<b><code>datalogger→get_dataStreams(v)</code></b>	Builds a list of all data streams hold by the data logger.
<b><code>datalogger→get_errorMessage()</code></b>	Returns the error message of the latest error with this function.
<b><code>datalogger→get_errorType()</code></b>	Returns the numerical error code of the latest error with this function.
<b><code>datalogger→get_functionDescriptor()</code></b>	Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.
<b><code>datalogger→get_hardwareId()</code></b>	Returns the unique hardware identifier of the function.
<b><code>datalogger→get_logicalName()</code></b>	Returns the logical name of the data logger.
<b><code>datalogger→get_measureNames()</code></b>	Returns the names of the measures recorded by the data logger.
<b><code>datalogger→get_module()</code></b>	Get the <code>YModule</code> object for the device on which the function is located.
<b><code>datalogger→get_module_async(callback, context)</code></b>	Get the <code>YModule</code> object for the device on which the function is located (asynchronous version).
<b><code>datalogger→get_oldestRunIndex()</code></b>	Returns the index of the oldest run for which the non-volatile memory still holds recorded data.
<b><code>datalogger→get_recording()</code></b>	Returns the current activation state of the data logger.
<b><code>datalogger→get_timeUTC()</code></b>	Returns the Unix timestamp for current UTC time, if known.
<b><code>datalogger→get_userData()</code></b>	Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<b><code>datalogger→isOnline()</code></b>	Checks if the function is currently reachable, without raising any error.
<b><code>datalogger→isOnline_async(callback, context)</code></b>	Checks if the function is currently reachable, without raising any error (asynchronous version).

**datalogger→load(msValidity)**

Preloads the function cache with a specified validity duration.

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

Preloads the function 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.

## yFindDataLogger()

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.

## yFirstDataLogger()

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()`**

Returns a descriptive text that identifies the function.

function **`describe()`**

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

### **`datalogger.forgetAllDataStreams()`**

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()`**

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()`**

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()`**

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

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

---

---

## **datalogger.get\_dataRun()**

Returns a data run object holding all measured data for a given period during which the module was turned on (a run).

function **get\_dataRun**( *runIdx* )

This object can then be used to retrieve measures (min, average and max) at a desired data rate.

### **Parameters :**

*runIdx* the index of the desired run

### **Returns :**

an YDataRun object

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

---

## **datalogger.get\_dataStreams()**

Builds a list of all data streams hold by the data logger.

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.

### **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()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

## **datalogger.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

## **datalogger.get\_dataloggerDescriptor()**

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_hardwareId()`**

Returns the unique hardware identifier of the function.

function `get_hardwareId( )`

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

## **`datalogger.get_logicalName()`**

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`.

---

Returns the names of the measures recorded by the data logger.

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.

---

## **`datalogger.get_module()`**

Get 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()`**

Get 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_oldestRunIndex()`**

Returns the index of the oldest run for which the non-volatile memory still holds recorded data.

function **`get_oldestRunIndex( )`**

**Returns :**

an integer corresponding to the index of the oldest run for which the non-volatile memory still holds recorded data

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

---

### **`datalogger.get_recording()`**

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()`**

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()`**

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()**

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

**function 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

---

## **datalogger.isOnline\_async()**

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.

---

## **datalogger.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

## **datalogger.load\_async()**

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 traffic 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.

---

## **`datalogger.nextDataLogger()`**

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()`**

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()`**

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()`**

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()**

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()**

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()**

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

## **3.7. 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:

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

#### 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()

Returns the average value of the measure observed at the specified time period.

function **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()

Returns the duration (in seconds) of the data run.

function **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()

Returns the maximal value of the measure observed at the specified time period.

```
function 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()**

Returns the names of the measures recorded by the data logger.

```
function 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()**

Returns the minimal value of the measure observed at the specified time period.

```
function 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`.

---

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()**

Returns the number of values accessible in this run, given the selected data samples interval.

```
function 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()**

Returns the number of seconds covered by each value in this run.

function **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()**

Changes the number of seconds covered by each value in this run.

function **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

## **3.8. Recorded data sequence**

DataStream objects represent a recorded measure sequence. They are returned by the data logger present on Yoctopuce sensors.

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

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

### **YDataStream methods**

**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\_dataSamplesInterval()**

Returns the number of seconds elapsed between two consecutive rows of this data 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 start time of the data stream, relative to the beginning of the run.

**datastream→get\_startTimeUTC()**

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

---

## **datastream.get\_columnCount()**

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()`.

This method fetches the whole data stream from the device, if not yet done.

**Returns :**

an unsigned number corresponding to the number of rows. On failure, throws an exception or returns zero.

---

## **datastream.get\_columnNames()**

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 archived streams created by summarizing a high-resolution data stream, there can be a suffix appended to the sensor identifier, such as `_min` for the minimum value, `_avg` for the average value and `_max` for the maximal value.

This method fetches the whole data stream from the device, if not yet done.

**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()**

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()**

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\_dataSamplesInterval()**

Returns the number of seconds elapsed between two consecutive rows of this data stream.

function **get\_dataSamplesInterval**( )

By default, the data logger records one row per second, but there might be alternative streams at lower resolution created by summarizing the original stream for archiving purposes.

This method does not cause any access to the device, as the value is preloaded in the object at instantiation time.

**Returns :**

an unsigned number corresponding to a number of seconds.

---

**datastream.get\_rowCount()**

Returns the number of data rows present in this stream.

function **get\_rowCount**( )

This method fetches the whole data stream from the device, if not yet done.

**Returns :**

an unsigned number corresponding to the number of rows. On failure, throws an exception or returns zero.

---

**datastream.get\_runIndex()**

Returns the run index of the data stream.

function **get\_runIndex**( )

A run can be made of multiple datastreams, for different time intervals. This method does not cause any access to the device, as the value is preloaded in the object at instantiation time.

**Returns :**

an unsigned number corresponding to the run index.

---

**datastream.get\_startTime()**

Returns the start time of the data stream, relative to the beginning of the run.

function **get\_startTime**( )

If you need an absolute time, use `get_startTimeUTC()`.

This method does not cause any access to the device, as the value is preloaded in the object at instantiation time.

**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()**

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.

This method does not cause any access to the device, as the value is preloaded in the object at instantiation time.

**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).

## 3.9. 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:

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

### 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 descriptive text that identifies the function.

**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 this function.

**dualpower→get\_errorType**()

Returns the numerical error code of the latest error with this function.

**dualpower→get\_extVoltage**()

Returns the measured voltage on the external power source, in millivolts.

**dualpower→get\_functionDescriptor**()

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

**dualpower→get\_hardwareId**()

Returns the unique hardware identifier of the function.

**dualpower→get\_logicalName**()

Returns the logical name of the power control.

**dualpower→get\_module**()

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

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

Get 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.

<b>dualpower→get_powerState()</b>
Returns the current power source for module functions that require lots of current.
<b>dualpower→get_userData()</b>
Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
<b>dualpower→isOnline()</b>
Checks if the function is currently reachable, without raising any error.
<b>dualpower→isOnline_async(callback, context)</b>
Checks if the function is currently reachable, without raising any error (asynchronous version).
<b>dualpower→load(msValidity)</b>
Preloads the function cache with a specified validity duration.
<b>dualpower→load_async(msValidity, callback, context)</b>
Preloads the function cache with a specified validity duration (asynchronous version).
<b>dualpower→nextDualPower()</b>
Continues the enumeration of dual power controls started using <code>yFirstDualPower()</code> .
<b>dualpower→registerValueCallback(callback)</b>
Registers the callback function that is invoked on every change of advertised value.
<b>dualpower→set_logicalName(newval)</b>
Changes the logical name of the power control.
<b>dualpower→set_powerControl(newval)</b>
Changes the selected power source for module functions that require lots of current.
<b>dualpower→set_userData(data)</b>
Stores a user context provided as argument in the userData attribute of the function.

## yFindDualPower()

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.

## yFirstDualPower()

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()**

Returns a descriptive text that identifies the function.

function **describe**( )

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

**dualpower.get\_advertisedValue()**

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()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

**dualpower.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

**dualpower.get\_extVoltage()**

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\_dualpowerDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function **get\_hardwareId**( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

## **dualpower.get\_logicalName()**

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()**

Get 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()**

Get 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()**

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()**

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()**

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()**

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

function `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

---

## dualpower.isOnline\_async()

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.

---

## dualpower.load()

Preloads the function 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 traffic for instance.

### Parameters :

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

### Returns :

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

---

## dualpower.load\_async()

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 traffic 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.

---

---

## **dualpower.nextDualPower()**

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()**

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()**

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()**

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()**

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

## 3.10. Yocto-hub port interface

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

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

### 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 descriptive text that identifies the function.

#### 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 this function.

#### hubport→get\_errorType()

Returns the numerical error code of the latest error with this function.

#### hubport→get\_functionDescriptor()

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

#### hubport→get\_hardwareId()

Returns the unique hardware identifier of the function.

#### hubport→get\_logicalName()

Returns the logical name of the Yocto-hub port, which is always the serial number of the connected module.

#### hubport→get\_module()

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

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

Get 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 function is currently reachable, without raising any error.

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

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

**hubport**→**load**(msValidity)

Preloads the function cache with a specified validity duration.

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

Preloads the function 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)

It is not possible to configure the logical name of a Yocto-hub port.

**hubport**→**set\_userData**(data)

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

## yFindHubPort()

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.

## yFirstHubPort()

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()

Returns a descriptive text that identifies the function.

function **describe**( )

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

## hubport.get\_advertisedValue()

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()

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 hit.

**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()

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()

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

---

## hubport.get\_errorType()

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

## hubport.get\_hubportDescriptor()

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\_hardwareId()

Returns the unique hardware identifier of the function.

function **get\_hardwareId**( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function. On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

## hubport.get\_logicalName()

Returns the logical name of the Yocto-hub port, which is always the serial number of the connected module.

function **get\_logicalName**( )

**Returns :**

a string corresponding to the logical name of the Yocto-hub port, which is always the serial number of the connected module

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

---

## hubport.get\_module()

Get 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()

Get 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()

Returns the current state of the Yocto-hub port.

function **get\_portState**( )

### Returns :

a value among `Y_PORTSTATE_OFF`, `Y_PORTSTATE_ON` and `Y_PORTSTATE_RUN` corresponding to the current state of the Yocto-hub port

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

---

## hubport.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.

---

## hubport.isOnline()

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

function **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

---

## hubport.isOnline\_async()

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.

---

## hubport.load()

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

## hubport.load\_async()

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 traffic 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.

---

## hubport.nextHubPort()

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()**

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()**

Changes the activation of the Yocto-hub port.

function **set\_enabled**( **newval** )

If the port is enabled, the \* connected module will be powered. Otherwise, port power will be 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()**

It is not possible to configure the logical name of a Yocto-hub port.

function **set\_logicalName**( **newval** )

The logical name is automatically set to the serial number of the connected module.

**Parameters :**

**newval** a string

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

## **hubport.set\_userData()**

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

## 3.11. 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:

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

### 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 descriptive text that identifies the function.

#### **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 measured value.

#### **humidity→get\_errorMessage()**

Returns the error message of the latest error with this function.

#### **humidity→get\_errorType()**

Returns the numerical error code of the latest error with this function.

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

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

#### **humidity→get\_hardwareId()**

Returns the unique hardware identifier of the function.

#### **humidity→get\_highestValue()**

Returns the maximal value observed.

#### **humidity→get\_logicalName()**

Returns the logical name of the humidity sensor.

#### **humidity→get\_lowestValue()**

Returns the minimal value observed.

#### **humidity→get\_module()**

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

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

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

<b>humidity→get_resolution()</b>
Returns the resolution of the measured values.
<b>humidity→get_unit()</b>
Returns the measuring unit for the measured value.
<b>humidity→get_userData()</b>
Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
<b>humidity→isOnline()</b>
Checks if the function is currently reachable, without raising any error.
<b>humidity→isOnline_async(callback, context)</b>
Checks if the function is currently reachable, without raising any error (asynchronous version).
<b>humidity→load(msValidity)</b>
Preloads the function cache with a specified validity duration.
<b>humidity→load_async(msValidity, callback, context)</b>
Preloads the function cache with a specified validity duration (asynchronous version).
<b>humidity→nextHumidity()</b>
Continues the enumeration of humidity sensors started using <code>yFirstHumidity()</code> .
<b>humidity→registerValueCallback(callback)</b>
Registers the callback function that is invoked on every change of advertised value.
<b>humidity→set_highestValue(newval)</b>
Changes the recorded maximal value observed.
<b>humidity→set_logicalName(newval)</b>
Changes the logical name of the humidity sensor.
<b>humidity→set_lowestValue(newval)</b>
Changes the recorded minimal value observed.
<b>humidity→set_userData(data)</b>
Stores a user context provided as argument in the userData attribute of the function.

## yFindHumidity()

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.



---

## yFirstHumidity()

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()

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 lineat interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

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

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

## humidity.describe()

Returns a descriptive text that identifies the function.

```
function describe( )
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

## humidity.get\_advertisedValue()

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()**

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()**

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`.

---

## **humidity.get\_errorMessage()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

## **humidity.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

## **humidity.get\_humidityDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

---

function **get\_hardwareId**( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

## **humidity.get\_highestValue()**

Returns the maximal value observed.

function **get\_highestValue**( )

**Returns :**

a floating point number corresponding to the maximal value observed

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

---

## **humidity.get\_logicalName()**

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()**

Returns the minimal value observed.

function **get\_lowestValue**( )

**Returns :**

a floating point number corresponding to the minimal value observed

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

---

## **humidity.get\_module()**

Get 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()**

Get 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\_resolution()**

Returns the resolution of the measured values.

function **get\_resolution()** ( )

The resolution corresponds to the numerical precision of the values, 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()**

Returns the measuring unit for the measured value.

function **get\_unit()** ( )

**Returns :**

a string corresponding to the measuring unit for the measured value

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

---

## **humidity.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.

---

## **humidity.isOnline()**

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

function **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

---

**humidity.isOnline\_async()**

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.

---

**humidity.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

**humidity.load\_async()**

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 traffic 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.

---

## **humidity.nextHumidity()**

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.registerValueCallback()**

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()**

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.

---

## **humidity.set\_logicalName()**

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()

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.

---

## humidity.set\_userData()

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

## 3.12. 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:

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

### 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 descriptive text that identifies the function.

#### 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 this function.

#### led→get\_errorType()

Returns the numerical error code of the latest error with this function.

#### led→get\_functionDescriptor()

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

#### led→get\_hardwareId()

Returns the unique hardware identifier of the function.

<b>led→get_logicalName()</b>	Returns the logical name of the led.
<b>led→get_luminosity()</b>	Returns the current led intensity (in per cent).
<b>led→get_module()</b>	Get the YModule object for the device on which the function is located.
<b>led→get_module_async(callback, context)</b>	Get the YModule object for the device on which the function is located (asynchronous version).
<b>led→get_power()</b>	Returns the current led state.
<b>led→get_userData()</b>	Returns the value of the userData attribute, as previously stored using method set_userData.
<b>led→isOnline()</b>	Checks if the function is currently reachable, without raising any error.
<b>led→isOnline_async(callback, context)</b>	Checks if the function is currently reachable, without raising any error (asynchronous version).
<b>led→load(msValidity)</b>	Preloads the function cache with a specified validity duration.
<b>led→load_async(msValidity, callback, context)</b>	Preloads the function cache with a specified validity duration (asynchronous version).
<b>led→nextLed()</b>	Continues the enumeration of leds started using yFirstLed().
<b>led→registerValueCallback(callback)</b>	Registers the callback function that is invoked on every change of advertised value.
<b>led→set_blinking(newval)</b>	Changes the current led signaling mode.
<b>led→set_logicalName(newval)</b>	Changes the logical name of the led.
<b>led→set_luminosity(newval)</b>	Changes the current led intensity (in per cent).
<b>led→set_power(newval)</b>	Changes the state of the led.
<b>led→set_userData(data)</b>	Stores a user context provided as argument in the userData attribute of the function.

## yFindLed()

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.

## **yFirstLed()**

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()**

Returns a descriptive text that identifies the function.

function **describe()**

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

## **led.get\_advertisedValue()**

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()**

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()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

## **led.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

## **led.get\_ledDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function **get\_hardwareId**( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function. On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

## **led.get\_logicalName()**

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()**

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()**

Get 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()**

Get 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()**

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()**

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()**

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

function **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

---

## **led.isOnline\_async()**

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.

---

## **led.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

## **led.load\_async()**

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 traffic 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.

---

## led.nextLed()

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()

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()

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()

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()**

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()**

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()**

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

## **3.13. 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:

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

## 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 descriptive text that identifies the function.

### **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 measured value.

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

Returns the error message of the latest error with this function.

### **lightsensor→get\_errorType()**

Returns the numerical error code of the latest error with this function.

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

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

### **lightsensor→get\_hardwareId()**

Returns the unique hardware identifier of the function.

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

Returns the maximal value observed.

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

Returns the logical name of the light sensor.

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

Returns the minimal value observed.

### **lightsensor→get\_module()**

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

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

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

### **lightsensor→get\_resolution()**

Returns the resolution of the measured values.

### **lightsensor→get\_unit()**

Returns the measuring unit for the measured value.

### **lightsensor→get\_userData()**

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

**lightsensor→isOnline()**

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

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

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

**lightsensor→load(msValidity)**

Preloads the function cache with a specified validity duration.

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

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

**lightsensor→nextLightSensor()**

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

**lightsensor→registerValueCallback(callback)**

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

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

Changes the recorded maximal value observed.

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

Changes the logical name of the light sensor.

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

Changes the recorded minimal value observed.

**lightsensor→set\_userData(data)**

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

**yFindLightSensor()**

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.

**yFirstLightSensor()**

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()**

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.

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

**lightsensor.calibrateFromPoints()**

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](mailto:support@yoctopuce.com).

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

**lightsensor.describe()**

Returns a descriptive text that identifies the function.

function **describe**( )

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

**lightsensor.get\_advertisedValue()**

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()**

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()**

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`.

---

### **lightsensor.get\_errorMessage()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

### **lightsensor.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

### **lightsensor.get\_lightsensorDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function `get_hardwareId( )`

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

**lightsensor.get\_highestValue()**

Returns the maximal value observed.

function `get_highestValue( )`

**Returns :**

a floating point number corresponding to the maximal value observed

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

---

**lightsensor.get\_logicalName()**

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()**

Returns the minimal value observed.

function `get_lowestValue( )`

**Returns :**

a floating point number corresponding to the minimal value observed

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

---

**lightsensor.get\_module()**

Get 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()

Get 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\_resolution()

Returns the resolution of the measured values.

function **get\_resolution**( )

The resolution corresponds to the numerical precision of the values, 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()

Returns the measuring unit for the measured value.

function **get\_unit**( )

### Returns :

a string corresponding to the measuring unit for the measured value

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

---

## lightsensor.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.

---

## lightsensor.isOnline()

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

function **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

---

## **lightsensor.isOnline\_async()**

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.

---

## **lightsensor.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

## **lightsensor.load\_async()**

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 traffic 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.

---

## **lightsensor.nextLightSensor()**

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.registerValueCallback()**

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()**

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.

---

## **lightsensor.set\_logicalName()**

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()

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.

---

### lightsensor.set\_userData()

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

## 3.14. 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:

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

#### 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**→**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.
<b>module</b> → <b>get_beacon()</b> Returns the state of the localization beacon.
<b>module</b> → <b>get_errorMessage()</b> Returns the error message of the last error with this module object.
<b>module</b> → <b>get_errorType()</b> Returns the numerical error code of the last error with this module object.
<b>module</b> → <b>get_firmwareRelease()</b> Returns the version of the firmware embedded in the module.
<b>module</b> → <b>get_functionDescriptor()</b> Returns a unique identifier of type <code>YFUN_DESCR</code> corresponding to the function.
<b>module</b> → <b>get_hardwareId()</b> Returns the unique hardware identifier of the module.
<b>module</b> → <b>get_icon2d()</b> Returns the icon of the module.
<b>module</b> → <b>get_logicalName()</b> Returns the logical name of the module.
<b>module</b> → <b>get_luminosity()</b> Returns the luminosity of the module informative leds (from 0 to 100).
<b>module</b> → <b>get_persistentSettings()</b> Returns the current state of persistent module settings.
<b>module</b> → <b>get_productId()</b> Returns the USB device identifier of the module.
<b>module</b> → <b>get_productName()</b> Returns the commercial name of the module, as set by the factory.
<b>module</b> → <b>get_productRelease()</b> Returns the hardware release version of the module.
<b>module</b> → <b>get_rebootCountdown()</b> Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.
<b>module</b> → <b>get_serialNumber()</b> Returns the serial number of the module, as set by the factory.
<b>module</b> → <b>get_upTime()</b> Returns the number of milliseconds spent since the module was powered on.
<b>module</b> → <b>get_usbBandwidth()</b> Returns the number of USB interfaces used by the module.
<b>module</b> → <b>get_usbCurrent()</b> Returns the current consumed by the module on the USB bus, in milli-amps.
<b>module</b> → <b>get_userData()</b> Returns the value of the <code>userData</code> attribute, as previously stored using method <code>set_userData</code> .
<b>module</b> → <b>isOnline()</b> Checks if the module is currently reachable, without raising any error.
<b>module</b> → <b>isOnline_async(callback, context)</b>



Checks if the module is currently reachable, without raising any error.
<b>module→load(msValidity)</b> Preloads the module cache with a specified validity duration.
<b>module→load_async(msValidity, callback, context)</b> Preloads the module cache with a specified validity duration (asynchronous version).
<b>module→nextModule()</b> Continues the module enumeration started using <code>yFirstModule()</code> .
<b>module→reboot(secBeforeReboot)</b> Schedules a simple module reboot after the given number of seconds.
<b>module→revertFromFlash()</b> Reloads the settings stored in the nonvolatile memory, as when the module is powered on.
<b>module→saveToFlash()</b> Saves current settings in the nonvolatile memory of the module.
<b>module→set_beacon(newval)</b> Turns on or off the module localization beacon.
<b>module→set_logicalName(newval)</b> Changes the logical name of the module.
<b>module→set_luminosity(newval)</b> Changes the luminosity of the module informative leds.
<b>module→set_usbBandwidth(newval)</b> Changes the number of USB interfaces used by the module.
<b>module→set_userData(data)</b> Stores a user context provided as argument in the <code>userData</code> attribute of the function.
<b>module→triggerFirmwareUpdate(secBeforeReboot)</b> Schedules a module reboot into special firmware update mode.

## yFindModule()

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.

## yFirstModule()

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()**

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.functionCount()**

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()**

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()**

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()**

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()**

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()**

Returns the error message of the last 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 last error message that occurred while using this module object

---

## **module.get\_errorType()**

Returns the numerical error code of the last 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 last error that occurred while using this module object

---

## **module.get\_firmwareRelease()**

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\_moduleDescriptor()**

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`.

---

## **module.get\_hardwareId()**

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

---

Returns the icon of the module.

The icon is a png image and does not exceeds 1024 bytes.

**Returns :**

a binary buffer with module icon, in png format.

---

## **module.get\_logicalName()**

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()**

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()**

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()**

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()**

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()**

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()**

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()**

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()**

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()**

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()**

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()**

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()**

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()**

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()

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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

## module.load\_async()

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 traffic 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()

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()**

Schedules a simple module reboot after the given number of seconds.

function **reboot**( **secBeforeReboot** )

### **Parameters :**

**secBeforeReboot** number of seconds before rebooting

### **Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

## **module.revertFromFlash()**

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

function **revertFromFlash**( )

### **Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

## **module.saveToFlash()**

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` if the call succeeds.

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

---

## **module.set\_beacon()**

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()**

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()**

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()**

Changes the number of USB interfaces used by the module.

function **set\_usbBandwidth**( **newval**)

**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()**

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()**

Schedules a module reboot into special firmware update mode.

function **triggerFirmwareUpdate**( **secBeforeReboot**)

---

**Parameters :**

**secBeforeReboot** number of seconds before rebooting

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

## 3.15. 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:

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

### 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 in to it.

#### network→describe()

Returns a descriptive text that identifies the function.

#### 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\_callbackMaxDelay()

Returns the maximum wait time between two callback notifications, in seconds.

#### network→get\_callbackMinDelay()

Returns the minimum wait time between two callback notifications, in seconds.

#### network→get\_callbackUrl()

Returns the callback URL to notify of significant state changes.

#### network→get\_errorMessage()

Returns the error message of the latest error with this function.

#### network→get\_errorType()

Returns the numerical error code of the latest error with this function.

#### network→get\_functionDescriptor()

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

#### network→get\_hardwareId()

Returns the unique hardware identifier of the function.

#### network→get\_ipAddress()

Returns the IP address currently in use by the device.

#### network→get\_logicalName()

Returns the logical name of the network interface, corresponding to the network name of the module.

**network→get\_macAddress()**

Returns the MAC address of the network interface.

**network→get\_module()**

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

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

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

**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→isOnline()**

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

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

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

**network→load(msValidity)**

Preloads the function cache with a specified validity duration.

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

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

**network→nextNetwork()**

Continues the enumeration of network interfaces started using yFirstNetwork().

**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\_callbackMaxDelay(newval)**

Changes the maximum wait time between two callback notifications, in seconds.

**network→set\_callbackMinDelay(newval)**

Changes the minimum wait time between two callback notifications, in seconds.

**network→set\_callbackUrl(newval)**

Changes the callback URL to notify of significant state changes.

**network→set\_logicalName(newval)**

Changes the logical name of the network interface, corresponding to the network name of the module.

**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→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.

## yFindNetwork()

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.

## yFirstNetwork()

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()

Connects to the notification callback and saves the credentials required to log in to it.

```
function callbackLogin( username, password)
```

The password will not be stored into the module, only a hashed copy of the credentials will be 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()

Returns a descriptive text that identifies the function.

```
function describe( )
```

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

### Returns :

a string that describes the function

---

## network.get\_adminPassword()

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()

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()

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\_callbackMaxDelay()**

Returns the maximum wait time between two callback notifications, in seconds.

function `get_callbackMaxDelay( )`

**Returns :**

an integer corresponding to the maximum wait time between two callback notifications, in seconds

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

---

**network.get\_callbackMinDelay()**

Returns the minimum wait time between two callback notifications, in seconds.

function `get_callbackMinDelay( )`

**Returns :**

an integer corresponding to the minimum wait time between two callback notifications, in seconds

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

---

**network.get\_callbackUrl()**

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\_errorMessage()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

**network.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

---

## network.get\_networkDescriptor()

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\_hardwareId()

Returns the unique hardware identifier of the function.

function `get_hardwareId( )`

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

## network.get\_ipAddress()

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()

Returns the logical name of the network interface, corresponding to the network name of the module.

function `get_logicalName( )`

**Returns :**

a string corresponding to the logical name of the network interface, corresponding to the network name of the module

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

---

## network.get\_macAddress()

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()**

Get 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()**

Get 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\_primaryDNS()**

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()**

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 on 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 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()**

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()**

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()**

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()**

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()**

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.isOnline()**

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

function **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

---

**network.isOnline\_async()**

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.

---

**network.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

**network.load\_async()**

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 traffic 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.

---

## network.nextNetwork()

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.registerValueCallback()

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()

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()

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 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\_callbackMaxDelay()

Changes the maximum wait time between two callback notifications, in seconds.

```
function set_callbackMaxDelay( newval)
```

**Parameters :**

**newval** an integer corresponding to the maximum wait 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\_callbackMinDelay()

Changes the minimum wait time between two callback notifications, in seconds.

```
function set_callbackMinDelay( newval)
```

**Parameters :**

**newval** an integer corresponding to the minimum wait 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()

Changes the callback URL to notify of 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 of significant state changes

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

## **network.set\_logicalName()**

Changes the logical name of the network interface, corresponding to the network 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 network interface, corresponding to the network name of the module

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

## **network.set\_primaryDNS()**

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 will override 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()**

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 will override 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()

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()

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.useDHCP()

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 will use 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()

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.

## 3.16. 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:

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

### 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 descriptive text that identifies the function.

#### 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 measured value.

#### pressure→get\_errorMessage()

Returns the error message of the latest error with this function.

#### pressure→get\_errorType()

Returns the numerical error code of the latest error with this function.

#### pressure→get\_functionDescriptor()

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

#### pressure→get\_hardwareId()

Returns the unique hardware identifier of the function.

#### pressure→get\_highestValue()

Returns the maximal value observed.

#### pressure→get\_logicalName()

Returns the logical name of the pressure sensor.

#### pressure→get\_lowestValue()

Returns the minimal value observed.

#### pressure→get\_module()

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

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

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

**pressure→get\_resolution()**

Returns the resolution of the measured values.

**pressure→get\_unit()**

Returns the measuring unit for the measured value.

**pressure→get\_userData()**

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

**pressure→isOnline()**

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

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

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

**pressure→load(msValidity)**

Preloads the function cache with a specified validity duration.

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

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

**pressure→nextPressure()**

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

**pressure→registerValueCallback(callback)**

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

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

Changes the recorded maximal value observed.

**pressure→set\_logicalName(newval)**

Changes the logical name of the pressure sensor.

**pressure→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**pressure→set\_userData(data)**

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

## yFindPressure()

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.

---

**yFirstPressure()**

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()**

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 lineat interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

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

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

**pressure.describe()**

Returns a descriptive text that identifies the function.

function **describe**( )

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

**pressure.get\_advertisedValue()**

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()**

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()**

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`.

---

### **pressure.get\_errorMessage()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

### **pressure.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

### **pressure.get\_pressureDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function **get\_hardwareId**( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function On failure, throws an exception or returns Y\_HARDWAREID\_INVALID.

---

## **pressure.get\_highestValue()**

Returns the maximal value observed.

function **get\_highestValue**( )

**Returns :**

a floating point number corresponding to the maximal value observed

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

---

## **pressure.get\_logicalName()**

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()**

Returns the minimal value observed.

function **get\_lowestValue**( )

**Returns :**

a floating point number corresponding to the minimal value observed

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

---

## **pressure.get\_module()**

Get 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()**

Get 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\_resolution()**

Returns the resolution of the measured values.

function **get\_resolution**( )

The resolution corresponds to the numerical precision of the values, 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()**

Returns the measuring unit for the measured value.

function **get\_unit**( )

**Returns :**

a string corresponding to the measuring unit for the measured value

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

---

## **pressure.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.

---

## **pressure.isOnline()**

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

function **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

---

## **pressure.isOnline\_async()**

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.

---

## **pressure.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

## **pressure.load\_async()**

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 traffic 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.

---

## **pressure.nextPressure()**

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.registerValueCallback()**

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()**

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.

---

## **pressure.set\_logicalName()**

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()**

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.

---

### **pressure.set\_userData()**

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

## **3.17. 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:

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

Global functions
<b>yFindRelay</b> ( <b>func</b> ) Retrieves a relay for a given identifier.
<b>yFirstRelay</b> () Starts the enumeration of relays currently accessible.
YRelay methods
<b>relay→describe</b> () Returns a descriptive text that identifies the function.
<b>relay→get_advertisedValue</b> () Returns the current value of the relay (no more than 6 characters).
<b>relay→get_errorMessage</b> () Returns the error message of the latest error with this function.
<b>relay→get_errorType</b> () Returns the numerical error code of the latest error with this function.
<b>relay→get_functionDescriptor</b> ()

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

**relay→get\_hardwareId()**

Returns the unique hardware identifier of the function.

**relay→get\_logicalName()**

Returns the logical name of the relay.

**relay→get\_module()**

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

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

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

**relay→get\_output()**

Returns the output state of the relay, when used as a simple switch (single throw).

**relay→get\_pulseTimer()**

Returns the number of milliseconds remaining before the relay is returned to idle position (state A), during a measured pulse generation.

**relay→get\_state()**

Returns the state of the relay (A for the idle position, B for the active position).

**relay→get\_userData()**

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

**relay→isOnline()**

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

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

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

**relay→load(msValidity)**

Preloads the function cache with a specified validity duration.

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

Preloads the function 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\_output(newval)**

Changes the output state of the relay, when used as a simple switch (single throw).

**relay→set\_state(newval)**

Changes the state of the relay (A for the idle position, B for the active position).

**relay→set\_userData(data)**

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

## **yFindRelay()**

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.

---

## **yFirstRelay()**

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.describe()**

Returns a descriptive text that identifies the function.

function **describe**( )

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

## **relay.get\_advertisedValue()**

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\_errorMessage()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

### **relay.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

### **relay.get\_relayDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function **get\_hardwareId**( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function. On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

### **relay.get\_logicalName()**

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\_module()**

Get 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()**

Get 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()**

Returns the output state of the relay, 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 relay, when used as a simple switch (single throw)

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

---

## **relay.get\_pulseTimer()**

Returns the number of milliseconds remaining before the relay 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 relay 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()**

Returns the state of the relay (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 relay (A for the idle position, B for the active position)

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

---

---

## relay.getUserData()

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

```
function getUserData( )
```

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

**Returns :**

the object stored previously by the caller.

---

## relay.isOnline()

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

```
function 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

---

## relay.isOnline\_async()

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.

---

## relay.load()

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

**relay.load\_async()**

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 traffic 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.

---

**relay.nextRelay()**

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()**

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 milliseconds

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

**relay.registerValueCallback()**

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()**

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\_output()**

Changes the output state of the relay, 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 relay, 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()**

Changes the state of the relay (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 relay (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\_userData()**

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

## 3.18. 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:

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

### 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 descriptive text that identifies the function.

#### **servo→get\_advertisedValue()**

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

#### **servo→get\_errorMessage()**

Returns the error message of the latest error with this function.

#### **servo→get\_errorType()**

Returns the numerical error code of the latest error with this function.

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

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

#### **servo→get\_hardwareId()**

Returns the unique hardware identifier of the function.

#### **servo→get\_logicalName()**

Returns the logical name of the servo.

#### **servo→get\_module()**

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

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

Get 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\_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 function is currently reachable, without raising any error.

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

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

**servo→load(msValidity)**

Preloads the function cache with a specified validity duration.

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

Preloads the function 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\_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\_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.

## yFindServo()

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.

## yFirstServo()

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()**

Returns a descriptive text that identifies the function.

function **describe()**

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

## **servo.get\_advertisedValue()**

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\_errorMessage()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

## **servo.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

## **servo.get\_servoDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function `get_hardwareId( )`

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

### **servo.get\_logicalName()**

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()**

Get 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()**

Get 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()**

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()**

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\_range()**

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()**

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()**

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

function **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

---

**servo.isOnline\_async()**

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.

---

## **servo.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

## **servo.load\_async()**

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 traffic 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.

---

## **servo.move()**

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()**

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()**

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\_logicalName()**

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()**

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()**

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\_range()**

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()**

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

## **3.19. 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:

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

#### **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 descriptive text that identifies the function.

**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 measured value.

**temperature→get\_errorMessage()**

Returns the error message of the latest error with this function.

**temperature→get\_errorType()**

Returns the numerical error code of the latest error with this function.

**temperature→get\_functionDescriptor()**

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

**temperature→get\_hardwareId()**

Returns the unique hardware identifier of the function.

**temperature→get\_highestValue()**

Returns the maximal value observed.

**temperature→get\_logicalName()**

Returns the logical name of the temperature sensor.

**temperature→get\_lowestValue()**

Returns the minimal value observed.

**temperature→get\_module()**

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

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

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

**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 measured value.

**temperature→get\_userData()**

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

**temperature→isOnline()**

Checks if the function is currently reachable, without raising any error.
<b>temperature→isOnline_async(callback, context)</b> Checks if the function is currently reachable, without raising any error (asynchronous version).
<b>temperature→load(msValidity)</b> Preloads the function cache with a specified validity duration.
<b>temperature→load_async(msValidity, callback, context)</b> Preloads the function cache with a specified validity duration (asynchronous version).
<b>temperature→nextTemperature()</b> Continues the enumeration of temperature sensors started using <code>yFirstTemperature()</code> .
<b>temperature→registerValueCallback(callback)</b> Registers the callback function that is invoked on every change of advertised value.
<b>temperature→set_highestValue(newval)</b> Changes the recorded maximal value observed.
<b>temperature→set_logicalName(newval)</b> Changes the logical name of the temperature sensor.
<b>temperature→set_lowestValue(newval)</b> Changes the recorded minimal value observed.
<b>temperature→set_sensorType(newval)</b> Modify the temperature sensor type.
<b>temperature→set_userData(data)</b> Stores a user context provided as argument in the <code>userData</code> attribute of the function.

## yFindTemperature()

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.

## yFirstTemperature()

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()**

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 lineat interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

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

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

## **temperature.describe()**

Returns a descriptive text that identifies the function.

function **describe**( )

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

## **temperature.get\_advertisedValue()**

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()**

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()**

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`.

---

**temperature.get\_errorMessage()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

**temperature.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

**temperature.get\_temperatureDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function `get_hardwareId( )`

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

---

## temperature.get\_highestValue()

Returns the maximal value observed.

function **get\_highestValue**( )

### Returns :

a floating point number corresponding to the maximal value observed

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

---

## temperature.get\_logicalName()

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()

Returns the minimal value observed.

function **get\_lowestValue**( )

### Returns :

a floating point number corresponding to the minimal value observed

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

---

## temperature.get\_module()

Get 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()

Get 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\_resolution()**

Returns the resolution of the measured values.

function **get\_resolution()** ( )

The resolution corresponds to the numerical precision of the values, 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()**

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` and `Y_SENSORTYPE_TYPE_T` corresponding to the temperature sensor type

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

---

### **temperature.get\_unit()**

Returns the measuring unit for the measured value.

function **get\_unit()** ( )

**Returns :**  
a string corresponding to the measuring unit for the measured value

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

---

### **temperature.getUserData()**

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

function **getUserData()** ( )

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

**Returns :**  
the object stored previously by the caller.

---

### **temperature.isOnline()**

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

function **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

---

## **temperature.isOnline\_async()**

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.

---

## **temperature.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

## **temperature.load\_async()**

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 traffic 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.

---

## **temperature.nextTemperature()**

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.registerValueCallback()**

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()**

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\_logicalName()**

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()

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\_sensorType()

Modify the temperature sensor type.

function **set\_sensorType**( **newval**)

This function is used to to define the type of thermo couple (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 and Y\_SENSORTYPE\_TYPE\_T

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

### temperature.set\_userData()

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

## 3.20. 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:

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

### 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 descriptive text that identifies the function.

**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 measured value.

**voltage→get\_errorMessage()**

Returns the error message of the latest error with this function.

**voltage→get\_errorType()**

Returns the numerical error code of the latest error with this function.

**voltage→get\_functionDescriptor()**

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

**voltage→get\_hardwareId()**

Returns the unique hardware identifier of the function.

**voltage→get\_highestValue()**

Returns the maximal value observed.

**voltage→get\_logicalName()**

Returns the logical name of the voltage sensor.

**voltage→get\_lowestValue()**

Returns the minimal value observed.

**voltage→get\_module()**

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

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

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

**voltage→get\_resolution()**

Returns the resolution of the measured values.

**voltage→get\_unit()**

Returns the measuring unit for the measured value.

**voltage→get\_userData()**

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

**voltage→isOnline()**

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

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



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

**voltage→load(msValidity)**

Preloads the function cache with a specified validity duration.

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

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

**voltage→nextVoltage()**

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

**voltage→registerValueCallback(callback)**

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

**voltage→set\_highestValue(newval)**

Changes the recorded maximal value observed.

**voltage→set\_logicalName(newval)**

Changes the logical name of the voltage sensor.

**voltage→set\_lowestValue(newval)**

Changes the recorded minimal value observed.

**voltage→set\_userData(data)**

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

## yFindVoltage()

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.

## yFirstVoltage()

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()**

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 lineat interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

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

### **Parameters :**

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

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

### **Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

## **voltage.describe()**

Returns a descriptive text that identifies the function.

function **describe**( )

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

### **Returns :**

a string that describes the function

---

## **voltage.get\_advertisedValue()**

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()**

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()**

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`.

---

## **voltage.get\_errorMessage()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

## **voltage.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

## **voltage.get\_voltageDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function **get\_hardwareId**( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

### **Returns :**

a string that uniquely identifies the function On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

## **voltage.get\_highestValue()**

Returns the maximal value observed.

function **get\_highestValue**( )

**Returns :**

a floating point number corresponding to the maximal value observed

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

---

**voltage.get\_logicalName()**

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()**

Returns the minimal value observed.

function **get\_lowestValue**( )

**Returns :**

a floating point number corresponding to the minimal value observed

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

---

**voltage.get\_module()**

Get 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()**

Get 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\_resolution()**

Returns the resolution of the measured values.

function **get\_resolution**( )

The resolution corresponds to the numerical precision of the values, 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()**

Returns the measuring unit for the measured value.

function **get\_unit**( )

### **Returns :**

a string corresponding to the measuring unit for the measured value

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

---

## **voltage.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.

---

## **voltage.isOnline()**

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

function **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

---

## **voltage.isOnline\_async()**

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.

---

## **voltage.load()**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

## **voltage.load\_async()**

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 traffic 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.

---

## **voltage.nextVoltage()**

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.registerValueCallback()**

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()**

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.

---

## **voltage.set\_logicalName()**

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()**

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.

## **voltage.set\_userData()**

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

## **3.21. 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:

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

### **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 descriptive text that identifies the function.

#### **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()**

Return true if external power supply voltage is too low.

#### **vsource→get\_failure()**

Return true if the module is in failure mode.

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

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

#### **vsource→get\_hardwareId()**

Returns the unique hardware identifier of the function.

#### **vsource→get\_logicalName()**

Returns the logical name of the voltage source.

#### **vsource→get\_module()**

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

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

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

#### **vsource→get\_overCurrent()**



Return true if the appliance connected to the device is too greedy .
<b>vsource→get_overHeat()</b> Return TRUE if the module is overheating.
<b>vsource→get_overLoad()</b> Return true if the device is not able to maintaint the requested voltage output .
<b>vsource→get_regulationFailure()</b> Return true if the voltage output is too high regarding the requested voltage .
<b>vsource→get_unit()</b> Returns the measuring unit for the voltage.
<b>vsource→get_userData()</b> Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
<b>vsource→get_voltage()</b> Returns the voltage output command (mV)
<b>vsource→isOnline()</b> Checks if the function is currently reachable, without raising any error.
<b>vsource→isOnline_async(callback, context)</b> Checks if the function is currently reachable, without raising any error (asynchronous version).
<b>vsource→load(msValidity)</b> Preloads the function cache with a specified validity duration.
<b>vsource→load_async(msValidity, callback, context)</b> Preloads the function cache with a specified validity duration (asynchronous version).
<b>vsource→nextVSource()</b> Continues the enumeration of voltage sources started using <code>yFirstVSource()</code> .
<b>vsource→pulse(voltage, ms_duration)</b> Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.
<b>vsource→registerValueCallback(callback)</b> Registers the callback function that is invoked on every change of advertised value.
<b>vsource→reset()</b> Resets the device Output.
<b>vsource→set_logicalName(newval)</b> Changes the logical name of the voltage source.
<b>vsource→set_userData(data)</b> Stores a user context provided as argument in the userData attribute of the function.
<b>vsource→set_voltage(newval)</b> Tunes the device output voltage (milliVolts).
<b>vsource→voltageMove(target, ms_duration)</b> Performs a smooth move at constant speed toward a given value.

## yFindVSource()

Retrieves a voltage source for a given identifier.

**function yFindVSource( 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()**

Starts the enumeration of voltage sources currently accessible.

function **yFirstVSource()**

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()**

Returns a descriptive text that identifies the function.

function **describe()**

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

## **vsource.get\_advertisedValue()**

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`.

---

## **vsource.get\_errorMessage()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

### **vsource.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

### **vsource.get\_extPowerFailure()**

Return true if external power supply voltage is too low.

function **get\_extPowerFailure**( )

**Returns :**

either `Y_EXTPOWERFAILURE_FALSE` or `Y_EXTPOWERFAILURE_TRUE`

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

---

### **vsource.get\_failure()**

Return true if the module is in failure mode.

function **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`

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

---

### **vsource.get\_vsourceDescriptor()**

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`.

---

### **vsource.get\_hardwareId()**

Returns the unique hardware identifier of the function.

function **get\_hardwareId**( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function. On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

### **`vsources.get_logicalName()`**

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`.

---

### **`vsources.get_module()`**

Get 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`

---

### **`vsources.get_module_async()`**

Get 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.

---

### **`vsources.get_overCurrent()`**

Return true if the appliance connected to the device is too greedy .

function `get_overCurrent()`

**Returns :**

either `Y_OVERCURRENT_FALSE` or `Y_OVERCURRENT_TRUE`

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

---

### **`vsources.get_overHeat()`**

Return TRUE if the module is overheating.

---

function **get\_overHeat**( )

**Returns :**

either Y\_OVERHEAT\_FALSE or Y\_OVERHEAT\_TRUE

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

---

**vsource.get\_overLoad()**

Return true if the device is not able to maintain the requested voltage output .

function **get\_overLoad**( )

**Returns :**

either Y\_OVERLOAD\_FALSE or Y\_OVERLOAD\_TRUE

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

---

**vsource.get\_regulationFailure()**

Return true if the voltage output is too high regarding the requested voltage .

function **get\_regulationFailure**( )

**Returns :**

either Y\_REGULATIONFAILURE\_FALSE or Y\_REGULATIONFAILURE\_TRUE

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

---

**vsource.get\_unit()**

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.

---

**vsource.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.

---

**vsource.get\_voltage()**

Returns the voltage output command (mV)

function **get\_voltage**( )

**Returns :**

an integer corresponding to the voltage output command (mV)

---

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

---

### **`vsource.isOnline()`**

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

`function 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()`**

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()`**

Preloads the function 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 traffic for instance.

**Parameters :**

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

**Returns :**

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

---

### **`vsource.load_async()`**

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 traffic 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()**

Continues the enumeration of voltage sources started using `yFirstVSource()`.

function **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()**

Sets device output to a specific voltage, for a specified duration, then brings it automatically to 0V.

function **pulse**( **voltage**, **ms\_duration**)

**Parameters :**

- voltage** pulse voltage, in millivolts
- ms\_duration** pulse duration, in milliseconds

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

---

## **vsource.registerValueCallback()**

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.

---

## **vsource.reset()**

Resets the device Output.

function **reset**( )

This function must be called after any error condition. After an error condition, voltage output will be set to none and cannot be changed until this function is called.

**Returns :**

YAPI\_SUCCESS if the call succeeds.

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

---

**vsource.set\_logicalName()**

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.

---

**vsource.set\_userData()**

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

---

**vsource.set\_voltage()**

Tunes the device output voltage (milliVolts).

function **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()**

Performs a smooth move at constant speed toward a given value.

function **voltageMove**( target, ms\_duration)

**Parameters :**



**target** new output value at end of transition, in millivolts.  
**ms\_duration** transition duration, in milliseconds

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

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

## 3.22. Wireless function interface

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

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

### 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 descriptive text that identifies the function.

#### wireless→get\_advertisedValue()

Returns the current value of the wireless lan interface (no more than 6 characters).

#### wireless→get\_channel()

Returns the 802.

#### wireless→get\_errorMessage()

Returns the error message of the latest error with this function.

#### wireless→get\_errorType()

Returns the numerical error code of the latest error with this function.

#### wireless→get\_functionDescriptor()

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

#### wireless→get\_hardwareId()

Returns the unique hardware identifier of the function.

#### wireless→get\_linkQuality()

Returns the link quality, expressed in per cents.

#### wireless→get\_logicalName()

Returns the logical name of the wireless lan interface.

#### wireless→get\_module()

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

#### wireless→get\_module\_async(callback, context)

Get 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).

<b>wireless→get_userdata()</b>
Returns the value of the userData attribute, as previously stored using method <code>set_userdata</code> .
<b>wireless→isOnline()</b>
Checks if the function is currently reachable, without raising any error.
<b>wireless→isOnline_async(callback, context)</b>
Checks if the function is currently reachable, without raising any error (asynchronous version).
<b>wireless→joinNetwork(ssid, securityKey)</b>
Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).
<b>wireless→load(msValidity)</b>
Preloads the function cache with a specified validity duration.
<b>wireless→load_async(msValidity, callback, context)</b>
Preloads the function cache with a specified validity duration (asynchronous version).
<b>wireless→nextWireless()</b>
Continues the enumeration of wireless lan interfaces started using <code>yFirstWireless()</code> .
<b>wireless→registerValueCallback(callback)</b>
Registers the callback function that is invoked on every change of advertised value.
<b>wireless→set_logicalName(newval)</b>
Changes the logical name of the wireless lan interface.
<b>wireless→set_userdata(data)</b>
Stores a user context provided as argument in the userData attribute of the function.

## yFindWireless()

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.

## yFirstWireless()

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()**

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 will be 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()**

Returns a descriptive text that identifies the function.

function **describe**( )

The text always includes the class name, and may include as well either the logical name of the function or its hardware identifier.

**Returns :**

a string that describes the function

---

## **wireless.get\_advertisedValue()**

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()**

Returns the 802.

function **get\_channel**( )

11 channel currently used, or 0 when the selected network has not been found.

**Returns :**

an integer corresponding to the 802

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

---

## **wireless.get\_errorMessage()**

Returns the error message of the latest error with this function.

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 occurred while using this function object

---

## **wireless.get\_errorType()**

Returns the numerical error code of the latest error with this function.

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 occurred while using this function object

---

## **wireless.get\_wirelessDescriptor()**

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\_hardwareId()**

Returns the unique hardware identifier of the function.

function **get\_hardwareId**( )

The unique hardware identifier is made of the device serial number and of the hardware identifier of the function.

**Returns :**

a string that uniquely identifies the function. On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

---

## **wireless.get\_linkQuality()**

Returns the link quality, expressed in per cents.

function **get\_linkQuality**( )

**Returns :**

an integer corresponding to the link quality, expressed in per cents

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

---

## **wireless.get\_logicalName()**

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\_module()**

Get 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()**

Get 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()**

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()**

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()**

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()**

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

function `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

---

### **wireless.isOnline\_async()**

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.

---

### **wireless.joinNetwork()**

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()

Preloads the function 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 traffic for instance.

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

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

---

## wireless.load\_async()

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 traffic 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.

---

## wireless.nextWireless()

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()

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()**

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()**

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



## 4. Index

A	
adhocNetwork	179
API	10
C	
calibrate	97
calibrateFromPoints	28
callbackLogin	117
CheckLogicalName	11
D	
describe	18
DisableExceptions	11
E	
EnableExceptions	11
EnableUSBHost	12
F	
FindAnButton	18
FindCarbonDioxide	27
FindColorLed	35
FindCurrent	43
FindDataLogger	52
FindDualPower	66
FindHubPort	73
FindHumidity	80
FindLed	88
FindLightSensor	96
FindModule	105
FindNetwork	116
FindPressure	128
FindRelay	136
FindServo	144
FindTemperature	152
FindVoltage	161
FindVSource	169
FindWireless	178
FirstAnButton	18
FirstCarbonDioxide	27
FirstColorLed	36
FirstCurrent	44
FirstDataLogger	52
FirstDualPower	66
FirstHubPort	73
FirstHumidity	81
FirstLed	89

FirstLightSensor	96
FirstModule	105
FirstNetwork	116
FirstPressure	129
FirstRelay	137
FirstServo	144
FirstTemperature	152
FirstVoltage	161
FirstVSource	170
FirstWireless	178
forgetAllDataStreams	53
FreeAPI	12
functionCount	106
functionId	106
functionName	106
functionValue	106
G	
get_adminPassword	117
get_advertisedValue	19
get_analogCalibration	19
get_autoStart	53
get_averageValue	60
get_baudRate	74
get_beacon	107
get_blinking	89
get_calibratedValue	19
get_calibrationMax	19
get_calibrationMin	19
get_callbackCredentials	117
get_callbackMaxDelay	118
get_callbackMinDelay	118
get_callbackUrl	118
get_channel	179
get_columnCount	63
get_columnNames	63
get_currentRawValue	28
get_currentRunIndex	53
get_currentValue	29
get_data	63
get_dataRows	63
get_dataRun	54
get_dataSamplesInterval	64
get_dataStreams	54
get_duration	60
get_enabled	74
get_errorMessage	19
get_errorType	20
get_extPowerFailure	171
get_extVoltage	67
get_failure	171
get_firmwareRelease	107
get_functionDescriptor	20
get_hardwareId	20
get_highestValue	29
get_hslColor	37
get_icon2d	108
get_ipAddress	119
get_isPressed	20
get_lastTimePressed	20
get_lastTimeReleased	21
get_linkQuality	180
get_logicalName	21
get_lowestValue	30
get_luminosity	90

get_macAddress	119
get_maxValue	60
get_measureNames	55
get_minValue	61
get_module	21
get_module_async	21
get_neutral	146
get_oldestRunIndex	56
get_output	139
get_overCurrent	172
get_overHeat	172
get_overLoad	173
get_persistentSettings	108
get_portState	76
get_position	147
get_power	91
get_powerControl	69
get_powerState	69
get_primaryDNS	120
get_productId	108
get_productName	109
get_productRelease	109
get_pulseTimer	139
get_range	147
get_rawValue	22
get_readiness	120
get_rebootCountdown	109
get_recording	56
get_regulationFailure	173
get_resolution	31
get_rgbColor	38
get_rgbColorAtPowerOn	38
get_router	121
get_rowCount	64
get_runIndex	64
get_secondaryDNS	121
get_security	181
get_sensitivity	22
get_sensorType	156
get_serialNumber	109
get_ssid	181
get_startTime	64
get_startTimeUTC	61
get_state	139
get_subnetMask	121
get_timeUTC	56
get_unit	31
get_upTime	109
get_usbBandwidth	110
get_usbCurrent	110
get_userData	22
get_userPassword	121
get_valueCount	61
get_valueInterval	61
get_voltage	173
GetAPIVersion	12
GetTickCount	12
H	
HandleEvents	12
hslMove	38
I	
InitAPI	13
isOnline	22
isOnline_async	22

J	
joinNetwork	182
L	
load	23
load_async	23
M	
move	148
N	
nextAnButton	23
nextCarbonDioxide	32
nextColorLed	40
nextCurrent	49
nextDataLogger	58
nextDualPower	71
nextHubPort	77
nextHumidity	86
nextLed	93
nextLightSensor	102
nextModule	111
nextNetwork	123
nextPressure	134
nextRelay	141
nextServo	149
nextTemperature	158
nextVoltage	166
nextVSource	175
nextWireless	183
P	
pulse	141
R	
reboot	112
RegisterDeviceArrivalCallback	13
RegisterDeviceRemovalCallback	13
RegisterHub	14
RegisterLogFunction	14
registerValueCallback	24
reset	175
revertFromFlash	112
rgbMove	40
S	
saveToFlash	112
set_adminPassword	123
set_analogCalibration	24
set_autoStart	58
set_beacon	112
set_blinking	93
set_calibrationMax	24
set_calibrationMin	24
set_callbackCredentials	124
set_callbackMaxDelay	124
set_callbackMinDelay	124
set_callbackUrl	124
set_enabled	78
set_highestValue	33
set_hslColor	41
set_logicalName	25
set_lowestValue	33
set_luminosity	94
set_neutral	149
set_output	142
set_position	150
set_power	94
set_powerControl	71
set_primaryDNS	125

set_range	150
set_recording	59
set_rgbColor	41
set_rgbColorAtPowerOn	41
set_secondaryDNS	125
set_sensitivity	25
set_sensorType	159
set_state	142
set_timeUTC	59
set_usbBandwidth	113
set_userData	25
set_userPassword	126
set_valueInterval	62
set_voltage	176
SetDelegate	14
SetTimeout	14
Sleep	15
T	
triggerFirmwareUpdate	113
U	
UnregisterHub	15
UpdateDeviceList	15
UpdateDeviceList_async	15
useDHCP	126
useStaticIP	126
V	
voltageMove	176
Y	
YAnButton	16
YCarbonDioxide	25
YColorLed	34
YCurrent	42
YDataLogger	50
YDataRun	59
YDataStream	62
YDualPower	65
YHubPort	72
YHumidity	79
YLed	87
YLightSensor	94
YModule	103
YNetwork	114
YPressure	127
YRelay	135
YServo	143
YTemperature	150
YVoltage	159
YVSource	168
YWireless	177